



Development Kit User's Guide

Java Card™ 3 Platform, Version 3.0.2

Classic 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, Mozilla, Netscape, Javadoc, 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, Mozilla, Netscape, Javadoc, 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 régis 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 réexportations 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 régi 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.



Adobe PostScript

Contents

Preface xix

Part I Setup, Samples and Tools

1. Introduction 3

Java Card 3 Platform Architecture 4

Java Card TCK 5

2. Developing Classic Edition Applications 7

Classic Applet Development Process 7

Classic Development Kit Tools 8

Using the Classic Tools 9

3. Installation 11

Prerequisites to Installing the Development Kit 11

Install and Setup the Development Kit 12

▼ Downloading the Development Kit 12

▼ Setting Up the System Variables 14

Installed Files and Directories 17

Contents of All Releases 17

Contents of the Source Release 18

Uninstalling the Development Kit 19

4. Running the Samples 21

General Procedures for Building and Running the Samples 21

Building and Running the Samples 22

Running the classic_applets Samples 24

HelloWorld Sample 25

▼ Run the HelloWorld Sample 25

Channels Sample 26

▼ Run the Channels Sample 27

Service Sample 28

▼ Run the Service Sample 28

Utility Sample 29

PIN Protection 29

Storage of Portfolio 29

Stock Trading 30

Get Information On a Stock 30

▼ Run the Utility Sample 30

Wallet Sample 31

▼ Run the Wallet Sample 31

ObjectDeletion Sample 32

PhotoCard Sample 36

RMIPurse Sample 38

▼ Run RMIPurse 38

SecureRMIPurse Sample 39

SignatureMessageRecovery Sample 42

Message Recovery Order of Operations 42

Sample Application 43

▼ Run SignatureMessageRecovery 44

Running the reference_apps Samples	45
Biometry Sample Application	45
How the Biometric API Works	47
Implementation Notes	49
▼ Run the Biometry Sample	49
JavaPurse Sample Application	50
▼ Run the JavaPurse Sample	51
JavaPurseCrypto Sample	52
▼ Run the JavaPurseCrypto Sample	52
Transit Sample	53
▼ Run the Transit Sample	54
 5. Converting and Exporting Java Class Files	57
Setting Java Compiler Options	58
Running the Converter	59
Using Delimiters with Command Line Options	62
Using a Command Configuration File	63
File Naming for the Converter	63
Input File Naming Conventions	63
Output File Naming Conventions	64
Verification of Input and Output Files	64
Creating a debug.msk Output File	65
Using Export Files	65
Specifying an Export Map	66
Viewing an Export File as Text	67
 6. Compatibility for Classic Applets	69
Generating Application Modules From Classic Applets	69
Running the Normalizer	69

normalize Subcommands	70
copyright Subcommand	71
help Subcommand	71
7. Working With CAP Files	73
CAP File v2.2.2 Manifest File Syntax	73
Sample Manifest File	75
Generating a CAP File From a Java Card Assembly File	76
Running capgen	76
Producing a Text Representation of a CAP File	77
Running capdump	77
8. Packaging and Deploying Your Application	79
Installer Components and Data Flow	80
Running scriptgen	81
Sending and Receiving APDUs	82
Running apdutool	82
apdutool Examples	84
Directing Output to the Console	84
Directing Output to a File	84
Using APDU Script Files	84
APDUScript Preprocessor Commands	86
Setting Default Applets	87
On-Card Installer Applet AID	87
Downloading CAP Files and Creating Applets	87
Downloading the CAP File	87
Creating an Applet Instance	88
On-card Installer APDU Protocol	88
APDU Types	89

APDU Responses to Installation Requests	93
A Sample APDU Script	96
Using the On-card Installer for Deletion	99
How to Send a Deletion Request	99
APDU Requests to Delete Packages and Applets	99
APDU Responses to Deletion Requests	101
On-Card Installer Limits	103
9. Using the Reference Implementation	105
Running the RI	106
Installer Mask	108
Obtaining Resource Consumption Statistics	108
Getting Resource Statistics With the PhotoCard Sample	108
RI Limits	110
Input and Output	111
Working With EEPROM Image Files	111
Input EEPROM Image File	112
Output EEPROM Image File	112
Same Input and Output EEPROM Image File	112
Different Input and Output EEPROM Image Files	112
The Default ROM Mask	113
10. Producing a Mask File from Java Card Assembly Files	115
Running maskgen	115
Order of Packages on the Command Line	117
Version Numbers for Processed Packages	117
maskgen Example	117
11. Building a Custom RI From Sources	119
Steps for Building a Custom RI	119

- ▼ Building the 32-Bit Custom RI 120
- ▼ Testing the 32-Bit Custom RI 121
- ▼ Building the 16-Bit Custom RI 121

12. Verifying CAP and Export Files 123

Verifying CAP Files 123

Running `verifycap` 124

Verifying Export Files 125

Running `verifyexp` 125

Verifying Binary Compatibility 126

Running `verifyrev` 127

Command Line Options for Off-Card Verifier Tools 127

Part II Programming With the Development Kit

13. Using Cryptography Extensions 131

Supported Cryptography Classes 132

Instantiating the Classes 134

14. Localizing With The Development Kit 135

Localization Support for Java Utilities 135

Localizing a Java Program to a New Locale 136

Localization Support for `cref` 137

15. Programming to the Java Card RMI Client-Side API 139

Remote Stub Object 139

Java Card RMI Client-Side API 140

Package `rmiclientlib` 141

Package `clientlib` 141

16. Working with APDU I/O 143

The APDU I/O API	143
APDU I/O Classes and Interfaces	143
Exceptions	144
Two-interface Card Simulation	145
Examples of Use	145
To Connect To a Simulator	145
To Establish a T=0 Connection To a Card	146
To Power Up And Power Down the Card	146
To Exchange APDUs	147
To Print the APDU	148
17. Programming for the Large Address Space	149
Programming Large Applications and Libraries	149
Handling a Package as a Separate Code Space	150
Storing Large Amounts of Data	150
Example: The photocard Demo Applet	151
A. Java Card Assembly Syntax Example	153
B. Additional Optional Ant Tasks	179
Location and Installation	179
▼ Installing the Ant Tasks	179
▼ Setting Up the Optional Ant Tasks	180
Library Dependencies	181
Ant Task Descriptions	182
APDUTool	183
Errors	183
Examples	184
CapDump	185
Errors	185

Examples	185
Capgen	186
Errors	186
Examples	186
Converter	188
Parameters Specified As Nested Elements	189
Examples	190
DeployCap	191
Errors and Return Codes	191
Examples	191
Exp2Text	193
Errors	193
Examples	193
Maskgen	195
Parameters Specified As Nested Elements	195
Examples	196
Scriptgen	198
Errors	198
Examples	198
VerifyCap	199
Parameters Specified As Nested Elements	199
VerifyExp	201
Parameters Specified As Nested Elements	201
Errors	201
Examples	202
VerifyRev	203
Parameters Specified As Nested Elements	203
Errors	203

Examples	204
Custom Types	204
AppletNameAID	204
Example	204
JCAInputFile	205
Examples	205
ExportFiles	205
Examples	205
NetBeans Software Integration	206
Glossary	207
Index	217

Figures

FIGURE 1-1	Classic Edition Architecture	4
FIGURE 1-2	Architecture of Connected Edition	5
FIGURE 2-3	Process for Classic Applet Development and Deployment	8
FIGURE 2-4	Java Card Platform Conversion	10
FIGURE 4-5	Biometric Sample Sequence Diagram	46
FIGURE 5-6	Calls Between Packages Go Through The Export Files	66
FIGURE 8-7	Installer Components	80
FIGURE 8-8	On-card Installer APDU Transmission Sequence	89
FIGURE 12-9	Verifying a CAP file	124
FIGURE 12-10	Verifying An Export File	125
FIGURE 12-11	Verifying Binary Compatibility Of Export Files	127

Code Examples

[CODE EXAMPLE 9-1](#) PhotoCard Sample Showing Resource Statistic Output 109

[CODE EXAMPLE 11-1](#) Expected Console Output When Building 32-Bit Custom RI 121

[CODE EXAMPLE 11-2](#) Expected Console Output When Building 16-Bit Custom RI 122

Tables

TABLE 3-1	Contents of All Releases	17
TABLE 3-2	Contents of the Source Release <code>src</code> Directory	18
TABLE 4-3	Authenticate User Command	40
TABLE 5-4	Converter Command Line Arguments	60
TABLE 5-5	<code>exp2text</code> Command Line Options	67
TABLE 6-6	<code>normalize</code> Subcommand Options	70
TABLE 7-7	Name:Value Pairs in the <code>MANIFEST.MF</code> File	74
TABLE 7-8	<code>capgen</code> Command Line Options	76
TABLE 8-9	<code>scriptgen</code> Command Line Options	81
TABLE 8-10	<code>apdutool</code> Command Line Options	83
TABLE 8-11	Supported APDU Script File Commands	85
TABLE 8-12	Set Default Applets on Different Logical Channels	87
TABLE 8-13	<code>Select</code> APDU Command	90
TABLE 8-14	<code>Response</code> APDU Command	90
TABLE 8-15	<code>CAP Begin</code> APDU Command	90
TABLE 8-16	<code>CAP End</code> APDU Command	91
TABLE 8-17	<code>Component ## Begin</code> APDU Command	91
TABLE 8-18	<code>Component ## End</code> APDU Command	91
TABLE 8-19	<code>Component ## Data</code> APDU Command	91
TABLE 8-20	<code>Create Applet</code> APDU Command	92

TABLE 8-21	Abort APDU Command	92
TABLE 8-22	APDU Responses to Installation Requests	93
TABLE 8-23	Delete Package Command	100
TABLE 8-24	Delete Package and Applets Command	100
TABLE 8-25	Delete Applet Command	101
TABLE 8-26	APDU Responses to Deletion Requests	101
TABLE 8-27	APDU Response Format	103
TABLE 9-28	Protocols Supported by RE Executables	106
TABLE 9-29	Case Sensitive Command Line Options for <code>cref.bat</code>	107
TABLE 10-30	Command Line Arguments for the maskgen Tool	116
TABLE 12-31	verifycap Command Line Arguments	124
TABLE 12-32	verifyexp Command Line Argument	126
TABLE 12-33	verifycap, verifyexp, verifyrev Command Line Options	128
TABLE 13-34	Algorithms Implemented by the Cryptography Classes	133
TABLE B-1	Library Dependencies	181
TABLE B-2	Parameters for APDUTool	183
TABLE B-3	Parameters for CapDump	185
TABLE B-4	Parameters for Capgen	186
TABLE B-5	Parameters for Converter	188
TABLE B-6	Parameters for DeployCap	191
TABLE B-7	Parameters for Exp2Text	193
TABLE B-8	Parameters for Maskgen	195
TABLE B-9	Parameters for Scriptgen	198
TABLE B-10	Parameters for VerifyCap	199
TABLE B-11	Parameters for VerifyExp	201
TABLE B-12	Parameters for VerifyRev	203
TABLE B-13	Parameters for AppletNameAID	204
TABLE B-14	Parameters for JCAInputFile	205

Preface

This document describes how to use the Java Card 3 Platform, Classic Edition, Development Kit version 3.0.2 to develop classic applets. The Java Card 3 Platform currently includes versions 3.0, 3.0.1, and 3.0.2 of various Java Card technology products. The latest platform specifications are version 3.0.1. The Classic Edition Platform is an update of the Java Card technology in the Platform 2.2.2 release. Classic applets are applet-based applications with the same capabilities as applets in previous versions of the Java Card platform.

In contrast, the Java Card 3 Platform, Connected Edition, contains a new architecture that enables developers to integrate smart cards within IP networks and web services architectures. In the Connected Edition development kit you can create extended applets and servlets to take advantage of those features. The Connected Edition also allows the creation of classic applets if you use the classic APIs. You can run both the Classic and Connected Edition development kits simultaneously.

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

The Java Card API is compatible with international standards such as ISO 7816, and industry-specific standards such as Europay, Master Card, and Visa (EMV).

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

Who Should Use This Document

This *Development Kit User's Guide* is written for developers who are creating classic applets using the *Application Programming Interface, Java Card Platform, Version 3.0.1, Classic Edition*, and also for developers who are considering creating a vendor-specific framework based on the Java Card specifications.

Before You Read This Document

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

You should also become familiar with the Java Card specifications. You can download the Java Card specifications bundle separately from the Sun Microsystems web site at <http://java.sun.com/products/javacard>.

How This Document 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.

Note – Throughout this document the base installation directory is referred to by the `JC_CLASSIC_HOME` replacement variable. The default value for this is `JCDK3.0.2_ClassicEdition` but it can be changed during the installation process.

Part I: [Setup, Samples and Tools](#)

[Chapter 1, Introduction](#), provides an overview of the development kit and tool use.

[Chapter 3, Installation](#), describes the prerequisites to and procedures for installing the development kit.

[Chapter 4, Running the Samples](#), describes sample applets that illustrate the use of the Java Card API. It also describes demonstration programs that illustrate very important scenarios of applet masking and post-manufacture installation.

[Chapter 5, Converting and Exporting Java Class Files](#), provides an overview of the Converter and how to run it. It also describes how to use export files, including the `exp2text` tool to view an export file in ASCII format.

[Chapter 6, Compatibility for Classic Applets](#), describes how to use the Normalizer tool to modify classic applets to run on both the Classic and Connected Editions.

[Chapter 7, Working With CAP Files](#), describes how to use CAP files and their manifest files. It also describes how to use the `capgen` utility and the `capdump` utility.

[Chapter 8, Packaging and Deploying Your Application](#), describes how to package and deploy applet applications to a smart card, including how to download and delete packages and create and delete applet instances using `scriptgen`, `apdutool`, and the on-card installer.

[Chapter 9, Using the Reference Implementation](#), describes how to use the runtime environment simulator for the Java Card platform (Java Card runtime environment or Java Card RE).

[Chapter 10, Producing a Mask File from Java Card Assembly Files](#), describes how to use the `maskgen` utility.

[Chapter 11, Building a Custom RI From Sources](#), describes how to use the source release to build a custom runtime environment.

[Chapter 12, Verifying CAP and Export Files](#), provides an overview of the off-card verifier tool and details of running it.

Part II: [Programming With the Development Kit](#)

[Chapter 13, Using Cryptography Extensions](#), describes the cryptography APIs provided with this release.

[Chapter 14, Localizing With The Development Kit](#), describes the localization support available for Java-based programs and tools, C-Based programs and Java Card platform RMI (Java Card RMI) sample applications and client framework.

[Chapter 15, Programming to the Java Card RMI Client-Side API](#), describes the reference implementation of the client-side Java Card Remote Method Invocation API (client-side Java Card RMI API). See the Javadoc™ tool generated API specification at `JC_CLASSIC_HOME\docs\rmiclient`.

[Chapter 16, Working with APDU I/O](#), describes the APDU I/O library that is used by components of the Java Card development kit and can also be used to develop client applications and platform simulators. See the Javadoc™ tool generated API specification at `JC_CLASSIC_HOME\docs\apduio`.

[Chapter 17, Programming for the Large Address Space](#), describes how the large address space is implemented for the Java Card RE reference implementation of the Java Card 3 Platform, Classic Edition. It also describes how your applications can get the most out of a large address space implementation.

[Appendix A, Java Card Assembly Syntax Example](#), describes the Java Card platform assembly output of the Converter using a commented example file.

[Appendix B, Additional Optional Ant Tasks](#), describes how to use the optional, unsupported Apache Ant tasks included in the Java Card development kit to streamline using the tools. Several command line tools are grouped together into Ant tasks to allow for their combined use.

[Glossary](#) lists key terms used in the Classic and Connected Editions.

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	% su Password:
<i>AaBbCc123</i>	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	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. To delete a file, type <code>rm filename</code> .

Note – Characters display differently depending on browser settings. If characters do not display correctly, change the character encoding in your browser to Unicode UTF-8.

Related Documentation

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, Classic Edition*
- *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*
- *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*
- *Java Card Platform, Version 3.0, White Paper*
- *Application Programming Notes, Java Card Platform, Version 3.0.1, Classic Edition*
- *Off-Card Verifier for the Java Card Platform White Paper*
- *Java Card Technology for Smart Cards* by Zhiqun Chen (Prentice Hall, 2000)
- *Java Card RMI Client Application Programming Interface*
(see the Javadoc™ tool generated API specification at `JC_CLASSIC_HOME\docs\rmiclient`)
- *ISO 7816 Specification Parts 1-6*
- *The Java Programming Language (Java Series), Fourth Edition* by Ken Arnold and James Gosling and David Holmes (Prentice Hall, August 27, 2005)
- *The Java Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Prentice Hall, 1999)

Accessing Sun Documentation Online

Access Java platform technical documentation on the web at the Java Developer Connection™ program web site at

<http://java.sun.com/reference/docs>

Documentation, Support, and Training

Sun Function	URL
Documentation	http://www.sun.com/documentation/
Support	http://www.sun.com/support/
Training	http://www.sun.com/training/

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

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments to <http://java.sun.com/docs/forms/sendusmail.html>.

Please include the title of the document with your feedback:

Development Kit User's Guide, Java Card 3 Platform, Classic Edition

You can also contact the project team by email at: jc3-ri-feedback@sun.com.

PART I Setup, Samples and Tools

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

Introduction

The Java Card 3 Platform 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. The Connected Edition development kit is not included in this Classic Edition development kit. You must download the Connected Edition development kit separately.

This document and this development kit applies only to the Classic Edition. References to components, such as the Java Card runtime environment (RE), refer to the component as it exists in the Classic Edition.

You can, however, develop classic applets using the Connected Edition development kit, then bring the classic applets back to this Classic Edition development kit to double check that you have a strictly classic applet. The Connected Edition, when used in conjunction with the NetBeans IDE delivers very useful tools to simplify development and debugging and is highly recommended as your main development environment. [Chapter 2](#) describes the development process in more detail.

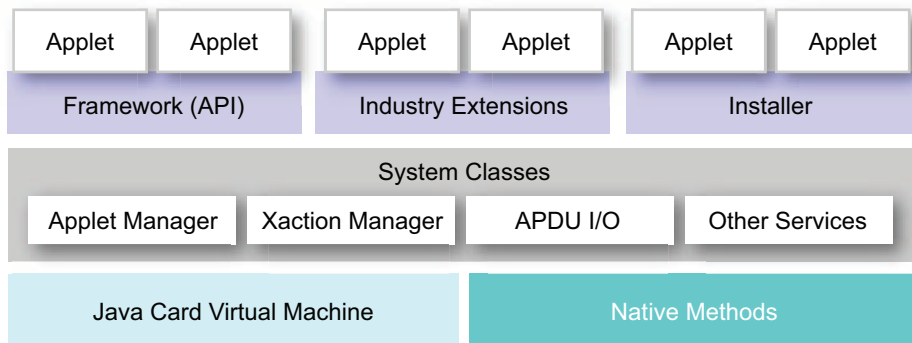
The Java Card development kit ships in binary-only bundles or bundles with both binary and source versions of the kit. In addition, cryptography extensions are available in some bundles. All sections in this document pertain to all these types of bundles, except where noted. For the contents in the bundles, see [“Contents of All Releases” on page 17](#). For more information on cryptography, see [Chapter 13](#).

Java Card 3 Platform Architecture

Any implementation of a Java Card runtime environment (Java Card RE) contains a virtual machine (VM) for the Java Card platform (Java Card virtual machine), the Java Card Application Programming Interface (API) classes, and support services.

The Classic Edition architecture illustrated in [FIGURE 1-1](#) is built on the classic Java Card VM, which is the same as the VM from previous releases of the Java Card development kit including version 2.2.2, 2.2.1, and so forth. Likewise, the classic APIs are very similar to the APIs from previous releases.

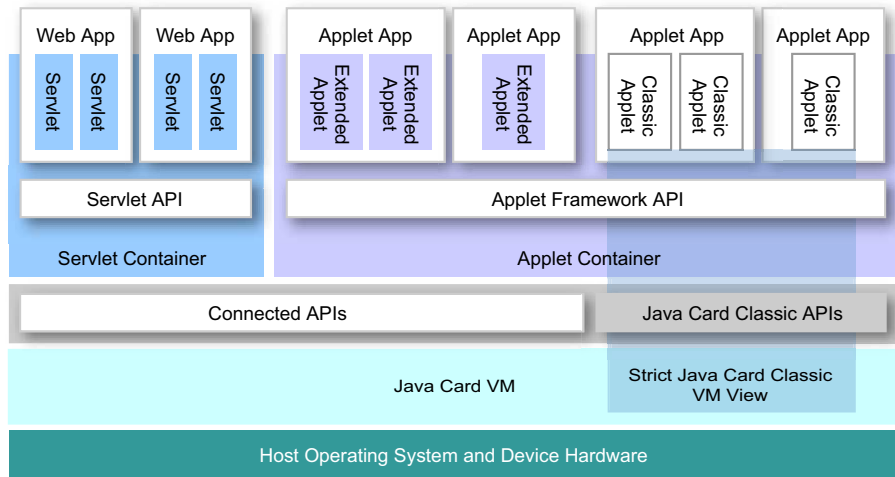
FIGURE 1-1 Classic Edition Architecture



In contrast, the Connected Edition is built on a very different Java Card VM, unlike the Classic Edition VM. The high-level architecture of the Connected Edition is illustrated in [FIGURE 1-2](#). Notice the classic APIs in a Connected Edition are built on smart cards that implement a view of the strictly classic Java Card VM, which supports only classic applet applications. This is why it is possible to use the Connected Edition development kit to develop classic applet applications.

However, the Connected Edition Java Card VM also supports extended applets and servlets. For more information on the Connected Edition, you must download a Connected Edition development kit separately, then see *Development Kit User's Guide, Java Card Platform, Version 3.0.1, Connected Edition*.

FIGURE 1-2 Architecture of Connected Edition



This classic development kit ships with a default Java Card RE that simulates a Java Card 3 Platform, Classic 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 `cref.bat`. The RI implements the ISO 7816-4:2005 specification, including support for up to twenty logical channels, as well as the extended APDU extensions as defined in ISO 7816-3. For more information on the RI, see [Chapter 9](#).

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 development kit 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.

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 applicable Java Card specification. To be in compliance, an implementation of the Java Card 3 platform, Classic Edition specification must pass the Java Card TCK 3.0.2 tests as described in *Java Card Technology Compatibility Kit, Version 3.0.2, Classic Edition User's Guide*.

Developing Classic Edition Applications

This chapter provides a brief description of the activities and development kit tools involved in developing applications for the Java Card 3 Platform, Classic Edition. See the *Application Programming Notes, Java Card Platform, Version 3.0.1, Classic Edition* for additional, advanced information not provided in this guide about creating applications for the Java Card 3 platform.

Classic Applet Development Process

Developing and debugging your classic applets can best be handled through the use of an IDE, such as the NetBeans IDE version 6.8. To do so, install the Connected development kit, then the NetBeans IDE. However, before you start developing any new classic applet applications, you might want to see [Chapter 4](#) describing the classic samples in this classic development kit and also read Part II of this book where classic program design issues are described.

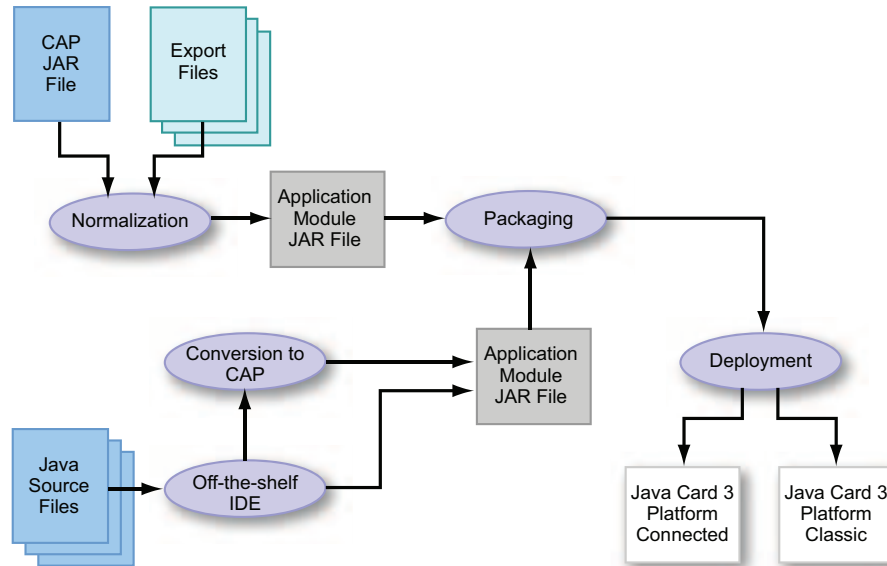
If you use the Connected Edition development kit to create your applet application, you can create your Java source code and debug it using the NetBeans IDE. You will also create a CAP file, which is a file distribution format that is a binary representation of a converted Java technology package. Then you will bring your classic applet application's CAP file back into this development kit. First, install this development kit as described in [Chapter 3](#). Next, starting with [Chapter 8, Packaging and Deploying Your Application](#), you can proceed onward in this book.

If you decide not to use the Connected Edition to develop your classic applet application, the process for developing a new classic applet application consists of first creating your Java code in the IDE of your choice, then debugging it. This version of the classic development kit does not include a tool for debugging your applet application.

When your Java source code is complete and you have compiled it to generate class files, you can then use the tools provided with the development kit to create CAP files that can be downloaded in the classic edition or connected edition simulators or cards. If you have existing CAP files in Java Card platform 2.x format, they can be converted to the 3.0.2 CAP file format through the normalization process.

FIGURE 2-3 shows the development and deployment process for classic applet applications using this development kit.

FIGURE 2-3 Process for Classic Applet Development and Deployment



Classic Development Kit Tools

The development kit for the Classic Edition consists of a suite of command line tools and samples for designing Java Card technology-based implementations and producing classic applet applications based on the *Application Programming Interface, Java Card Platform, Version 3.0.1, Classic Edition*.

Using the development kit's suite of tools is described in [“Using the Classic Tools” on page 9](#) and in more detail throughout this book.

- **apdutool** - A client-side tool used to send APDU commands to the RE and your on-card applet application. During the application deployment process, it can be used to read the output script file generated by `scriptgen` to send it to the Card Manager application, see [Chapter 8](#).
- **capdump** - Creates an ASCII version of a CAP file, see [Chapter 7](#).
- **capgen** - Generates a CAP file from a Java Card Assembly file, see [Chapter 7](#).
- **Converter** - Converts Java classes into a CAP file, a Java Card Assembly file, or an export file, see [Chapter 5](#).
- **cref** - Runs the RI from the command line, see [Chapter 9](#). There are three versions of `cref` to handle various communication protocols.
- **exp2text** - Allows you to view any export file in text format, see [Chapter 5](#).
- **on-card installer** - The on-card installer resides on the smart card and downloads Java Card technology packages to a smart card. It can also delete packages and applets, see [Chapter 8](#).
- **maskgen** - Produces a mask file from a set of Java Card Assembly files, see [Chapter 10](#).
- **Normalizer** - Enables classic applets to run on smart cards enabled with the Connected Edition and Classic Edition, see [Chapter 6](#).
- **off-card verifier** - Verifies the contents of a smart card using `verifycap`, `verifyexp`, and `verifyrev`, see [Chapter 12](#).
- **scriptgen** - The off-card installer, of which `scriptgen` is a part, resides on the desktop and generates script files for `apdutool`'s use, see [Chapter 8](#). If you have developed your classic applet application using the Connected Edition of the development kit and are now bringing your finished CAP file back into this classic development kit for packaging and deployment, the `scriptgen` tool described in this chapter is where you need to start.
- **optional Ant tasks** - Additional, optional, and unsupported Ant tasks that can streamline development by combining the command line tools into useful groups of tasks, see [Appendix B](#).

Using the Classic Tools

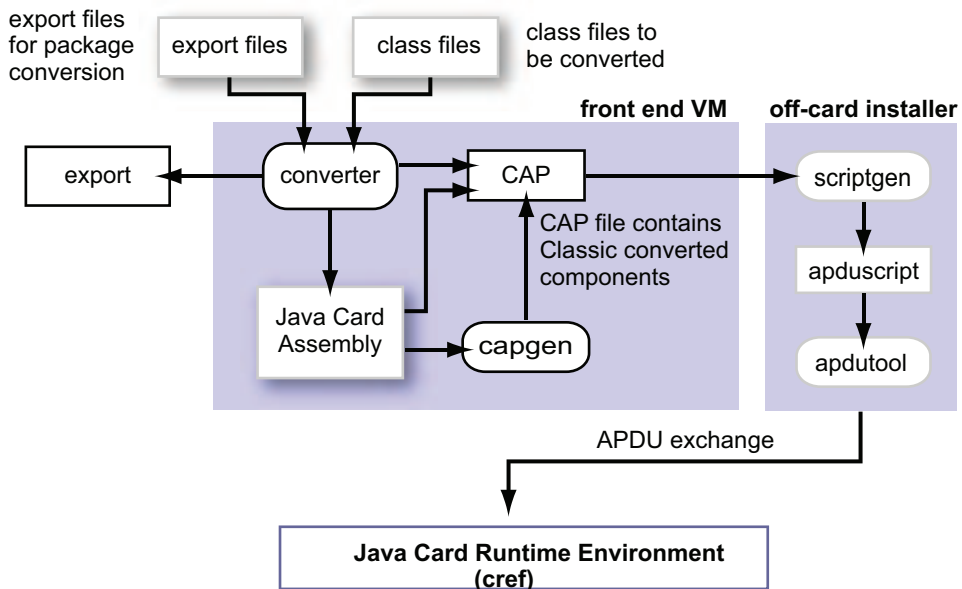
Each classic development kit tool performs a necessary function in producing a classic applet application for the Java Card 3 Platform, Classic Edition, see [FIGURE 2-4](#). An off-the-shelf IDE such as the NetBeans IDE can be used to write and debug your Java classes. Then, the tools in this development kit can be used to package and deploy your classic applet application.

Once you have written your Java programming language source code, it can be converted into a CAP file, packaged, and then sent via APDUs to a Java Card technology-enabled smart card. The data flow starts with Java programming language source being compiled with an IDE and then input to the Converter. The Converter tool can convert classes and any export files that comprise a Java package into a converted applet (CAP) file or into a Java Card technology-based Assembly (Java Card Assembly) file.

A CAP file is a binary representation of converted Java technology package. A Java Card Assembly file is a human-readable text representation of a converted package that you can use to aid testing and debugging. A Java Card Assembly file can also be used as input to the `capgen` tool to create a CAP file.

CAP files are processed, or “packaged,” by an off-card installer, `scriptgen`. This produces an APDU script file as input for the deployment process handled by `apdutool`, which then sends APDUs to the off-card installer on a smart card containing a Java Card RE implementation. The RI can be used as an emulator for a smart card environment.

FIGURE 2-4 Java Card Platform Conversion



Not shown in [FIGURE 2-4](#) is the tool `capdump`, which produces a simple ASCII version of the CAP file to aid in debugging. Also, the off-card verification tools are not shown.

Installation

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

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

Each development kit is provided in an executable JAR file bundle. See [Chapter 1](#) for a description of this development kit bundle and “[Contents of All Releases](#)” on [page 17](#) for a list of all the files installed by this development kit.

Note – The Java Card specifications are not included in the development kit. The specifications must be downloaded separately.

Prerequisites to Installing the Development Kit

The following software must be installed before installing a development kit:

- **Java Development Kit** - The commercial version of Java Development Kit (JDK™ software) version 6 Update 10 (JDK 6 Update 10) or later is required.

Download the JDK software from <http://java.sun.com/javase/downloads> and install it according to the instructions on the web site.

- **Apache ANT** - Apache Ant 1.6.5 or higher is required to run the samples from command line or to build the cref from source code.

Download and install Apache Ant version 1.6.5 or higher from <http://ant.apache.org>.

- **GCC compiler** - Minimal GNU for Windows (MinGW), version 5.1.4 or later is required to build the `crcf` and tools from sources.

Download MinGW from <http://sourceforge.net/projects/mingw>. For MinGW installation information, go to <http://www.mingw.org>.

- **NetBeans IDE (optional)** - The NetBeans IDE version 6.8 can be used to develop and debug classic applet applications. Download NetBeans IDE 6.8 from the following URL and install it according to the instructions on the web site:

<http://www.netbeans.org/>

- **Firefox browser (optional)** - The Firefox browser (not Internet Explorer) is considered as a trusted agent for running the RI. Firefox can be obtained from <http://www.mozilla.com>.

- **javax.comm package** - Install this if you are planning to use the development kit to communicate with a TLP224-compatible card reader.

Use the `javax.comm` package included in the latest version of the Java Communications API, available on Sun's web site at: <http://java.sun.com/products/javacomm>.

Follow the instructions provided in the file `Readme.html` to install the package, and make sure that the `comm.jar` file is added to the `CLASSPATH`.

Install and Setup the Development Kit

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

▼ Downloading the Development Kit

1. **Verify that JDK 6 Update 10 or later 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 JDK.

2. **Download the development kit JAR file to a directory of your choice.**

3. Launch the development kit installer.

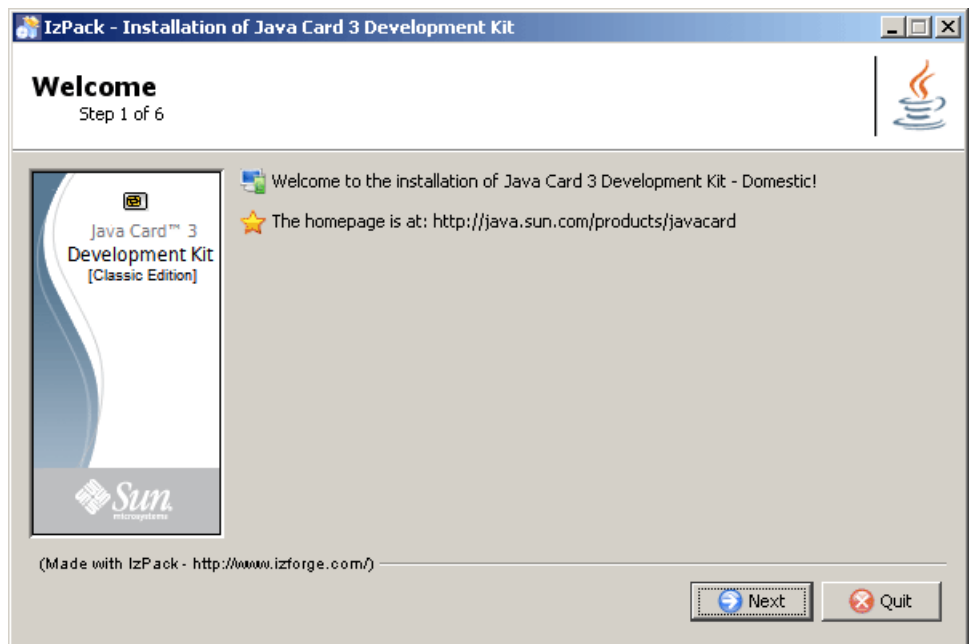
The development kit can be launched automatically when you download the JAR file or by using the Windows file manager tool to navigate to the directory containing the development kit JAR file and double clicking the file name or icon.

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.

During installation, the development kit is installed in C : \ JCDK3.0.2_ClassicEdition by default. 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 C:\JCDK3.0.2_ClassicEdition directory or the alternate installation directory you specify, is referred to as *JC_CLASSIC_HOME* throughout this document.

5. Click the Finish button to complete installation.

The bundle installs files and directories containing the binary files and source code as described in “[Contents of All Releases](#)” on [page 17](#). The files and directories are installed under the root installation directory referred to as *JC_CLASSIC_HOME* in this document.

▼ Setting Up the System Variables

1. Set a JAVA_HOME system variable to the JDK software root directory and put its bin\ in the PATH.

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:

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

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

Note – The GUI entrance to setting environment variables might differ depending on the Microsoft Windows operating system (OS) version in use. If necessary, see the OS online help.

- b. Alternately, to temporarily set JAVA_HOME, enter the following command in a command prompt window:**

```
set JAVA_HOME=java_home_path
```

For example, if the JDK software is stored in the c:\jdk6 directory, enter:

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

- c. After performing the previous steps, add JAVA_HOME\bin to the PATH:**

```
set PATH=%JAVA_HOME%\bin;%PATH%
```

This can also be set permanently in the Control Panel System settings.

2. Set a ANT_HOME system variable to the Ant root directory and put its bin\ in the PATH.

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:

- a. 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 root folder. For example, in the System variables box enter the following:**

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

Note – The GUI entrance to setting environment variables might differ depending on the Microsoft Windows operating system (OS) version in use. If necessary, see the OS online help.

- b. Alternately, to temporarily set ANT_HOME, enter the following command in a command prompt window:**

```
set ANT_HOME=ant_home_path
```

For example, if the ANT software is stored in the C:\ant\apache-ant1.6.5 directory, enter:

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

- c. After performing the previous steps, add ANT_HOME\bin to the PATH:**

```
set PATH=%ANT_HOME%\bin;%PATH%
```

This can also be set permanently in the Control Panel System settings.

3. Set a JC_CLASSIC_HOME system variable to the development kit root directory and add it to the PATH.

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

Note – The command line tools and included application samples require that the JC_CLASSIC_HOME variable is set correctly.

- a. To permanently set `JC_CLASSIC_HOME`, go to Windows Control Panel > System > Advanced > Environment Variables dialog and either create or edit a system variable named `JC_CLASSIC_HOME` so that its value is either `C:\JCDK3.0.2_ClassicEdition` or the directory you specified during installation. For example, in the System variables box enter the following:

Variable	Value
<code>JC_CLASSIC_HOME</code>	<code>C:\JCDK3.0.2_ClassicEdition</code>

Note – The GUI entrance to setting environment variables might differ depending on the Microsoft Windows operating system (OS) version in use. If necessary, see the OS online help.

- b. Alternately, to temporarily set `JC_CLASSIC_HOME`, enter the following command in a command prompt window:

```
set JC_CLASSIC_HOME=jc-home-path
```

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

```
set JC_CLASSIC_HOME=C:\JCDK3.0.2_ClassicEdition
```

- c. After performing the previous steps, add `JC_CLASSIC_HOME` to the `PATH`:

```
set PATH=%JC_CLASSIC_HOME%;%PATH%
```

This can also be set permanently in the Control Panel System settings.

4. 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 Files and Directories

The files and directories are installed under the root installation directory, either `C:\JCCK3.0.2_ClassicEdition` or the directory you specified during installation. The root installation directory is referred to as `JC_CLASSIC_HOME` in this guide.

The source release contains all the files installed with the binary release, plus a `src` directory.

Contents of All Releases

[TABLE 3-1](#) describes the files and directories that the installation procedure places in the root installation directory, represented by `JC_CLASSIC_HOME`.

These files are installed in binary releases and are also installed for source releases, see [TABLE 3-2](#).

TABLE 3-1 Contents of All Releases

Directory/File	Description
<code>api_export_files</code>	Contains the export files for version 3.0.2 of the Java Card API packages. If you have a development kit that includes cryptography, this also includes the directory <code>api_export_files\javacardx\crypto</code> .
<code>bin</code>	Contains all shell scripts or batch files for running the tools (such as the <code>apdutool</code> , <code>capdump</code> , <code>converter</code> and so forth), and the <code>crcf</code> binary executables.
<code>docs</code>	Contains the RI's APIs in Javadoc tool files and this book in both PDF and HTML format. It also contains two subdirectories each with respective compilations of the Javadoc tool files for the Java Card Client RMI API and the APDU I/O API. Note - The RI for the Classic Edition supports RMI but the RI for the Connected Edition does not.
<code>legal</code>	Contains license files.
<code>lib</code>	Contains all Java programming language JAR files required for the running tools using the BAT files provided in the <code>bin</code> directory.

TABLE 3-1 Contents of All Releases (*Continued*)

Directory/File	Description
samples	Contains sample applets.
RELEASENOTES	Release notes and copyright files for the development kit.
Uninstaller	Contains <code>uninstaller.jar</code> to run for uninstalling the product.

Contents of the Source Release

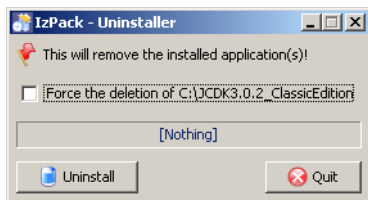
[TABLE 3-2](#) describes the items in the `src` directory installed if you have the source release of the development kit. For the descriptions of other installed items, see [“Contents of All Releases” on page 17](#).

TABLE 3-2 Contents of the Source Release `src` Directory

Directory/File	Description
api	Contains the source files for the development kit. Contains all of the <code>.java</code> files required to build a custom RI. If a new package must be added, it is added under this folder. If you have a development kit that includes cryptography, this also includes the directory <code>com/sun/javacard/crypto</code> .
installer	Contains the source files for the installer.
tools	Contains the source files for the development kit 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.
vm/c	Contains the source files of core VM.
vm/h	Contains the header files of core VM.
build.xml	The main file used to build the tools and <code>cref</code> in a single step.
apiImpl.jar	
vm_build.xml	

Uninstalling the Development Kit

To uninstall the development kit, version 3.0.2, run the Uninstaller tool found at *JC_CLASSIC_HOME*\Uninstaller\uninstaller.jar. Do not change the location of this tool. Before running the Uninstaller, exit all development kit tools and the NetBeans IDE. Any files under the control of the host operating system are not removed using the Uninstaller. When you execute the file, it displays this dialog:



Selecting the check box or not in this dialog box yields the same result, because in either case the Uninstaller removes the version 3.0.2 development kit in which the *uninstaller.jar* file resides.

You can also uninstall a development kit for any Java Card 3 Platform release by simply deleting all its *JC_CLASSIC_HOME* directories and files from your hard drive.

Running the Samples

The `samples` directory under `JC_CLASSIC_HOME` contains classic applet applications and reference applications that demonstrate the features of the Java Card 3, Classic Edition API. The reference application samples are blue print-like applications that demonstrate the interactions between various applications on the card using advanced features such as SIO and events.

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

- [General Procedures for Building and Running the Samples](#)
- [Running the `classic_applets` Samples](#)
- [Running the `reference_apps` Samples](#)

General Procedures for Building and Running the Samples

This section contains the following general procedures that developers use to build and run a sample. By default, the build script in the binary release produces a 32-bit version of `crcf` that supports dual interfaces of T=CL and T=1 protocols.

Each sample has a `build.xml` in its applet folder and, if applicable, `client` folder. If changes are made to a sample source file, developers can quickly test their changes by using `build.xml` with the `ant` tool to build a sample without running it. The `ant` tool uses the `build.xml` and the Development Kit tools to compile the Java programming language sources, convert the Java programming language class files, generate the APDU script files, and build the sample.

The `ant` tool runs the tools required to build the sample and generates the required output files. It displays a build status message at completion of the task.

See Part II and *Programming Notes, Java Card 3 Platform, Classic Edition* for information about creating classic applets.

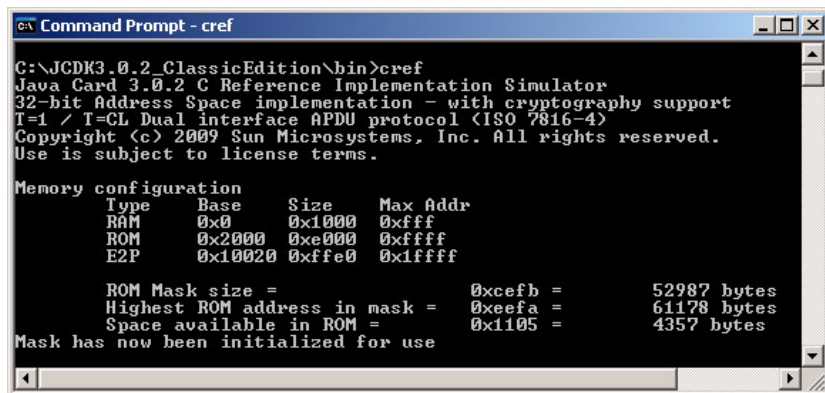
Building and Running the Samples

Before building and running a sample, verify that `ant` can be run from the command line and that `JC_CLASSIC_HOME` is set to the Development Kit installation directory in the Environment Variables dialog. “[Contents of All Releases](#)” on page 17 identifies the directory in which the Development Kit was installed. See “[Contents of All Releases](#)” on page 17 for installation and configuration instructions.

Three general actions are performed when running a sample:

1. In a Command Prompt window, start the RI by using the `cref` command with the options specified by the sample.

The following is an example of the output in the Command Prompt window when using the `cref` command to start the RI.



```
Command Prompt - cref
C:\JCDK3.0.2_ClassicEdition\bin>cref
Java Card 3.0.2 C Reference Implementation Simulator
32-bit Address Space implementation - with cryptography support
I=1 / T=CL Dual interface APDU protocol (ISO 7816-4)
Copyright (c) 2009 Sun Microsystems, Inc. All rights reserved.
Use is subject to license terms.

Memory configuration
  Type   Base      Size      Max Addr
  RAM    0x0         0x1000    0xffff
  ROM    0x2000     0xe000    0xffff
  E2P     0x10020     0xffe0    0x1ffff

  ROM Mask size =          0xcefb =      52987 bytes
  Highest ROM address in mask = 0xeefa =     61178 bytes
  Space available in ROM =    0x1105 =      4357 bytes
Mask has now been initialized for use
```

See [Chapter 9](#) for more information about using `cref` and its command line options.

2. In a second Command Prompt window, from the sample directory containing the appropriate `build.xml` file run the `ant` command with the appropriate target:

`ant target`

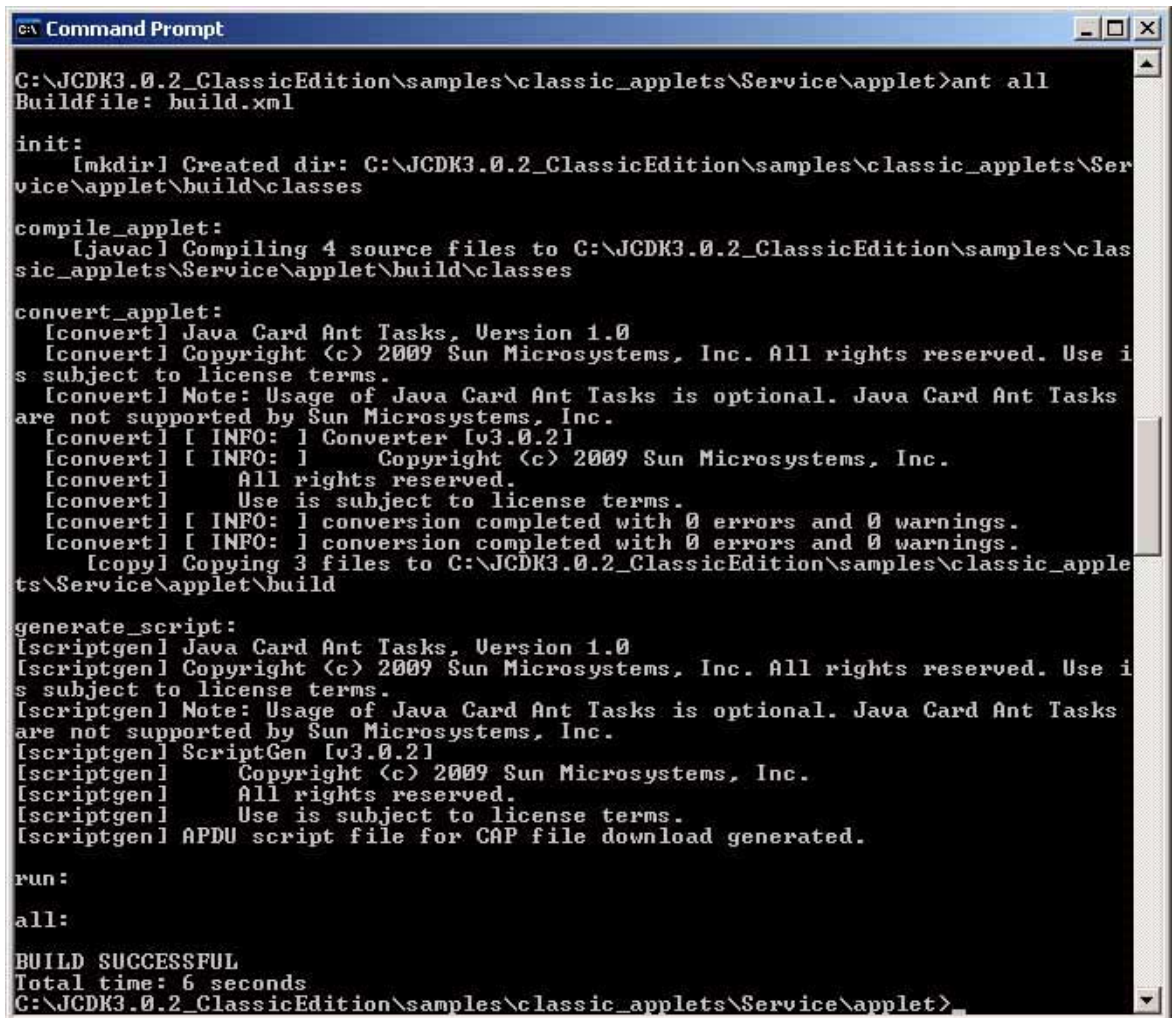
In the command, *target* represents the run option (such as `all` or `run1-1`) specified in the procedures for running the sample. Each sample might use one or more targets to run specific APDU scripts or multiple parts of the sample applet. The required targets are described in the procedures used to run an individual sample

With the exception of the Transit, RMIPurse, and SecureRMIPurse samples, a custom name can be specified for the output file generated by the ant command. Use the following command syntax to specify a custom name for the output file:

```
ant -Dredirect.output=outputfile_name target
```

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

The following is an example of the output in the Command Prompt window when running the Service applet.



```
C:\JCDK3.0.2_ClassicEdition\samples\classic_applets\Service\applet>ant all
Buildfile: build.xml

init:
[mkdir] Created dir: C:\JCDK3.0.2_ClassicEdition\samples\classic_applets\Service\applet\build\classes

compile_applet:
[javac] Compiling 4 source files to C:\JCDK3.0.2_ClassicEdition\samples\classic_applets\Service\applet\build\classes

convert_applet:
[convert] Java Card Ant Tasks, Version 1.0
[convert] Copyright (c) 2009 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
[convert] Note: Usage of Java Card Ant Tasks is optional. Java Card Ant Tasks are not supported by Sun Microsystems, Inc.
[convert] [ INFO: ] Converter [v3.0.2]
[convert] [ INFO: ] Copyright (c) 2009 Sun Microsystems, Inc.
[convert] All rights reserved.
[convert] Use is subject to license terms.
[convert] [ INFO: ] conversion completed with 0 errors and 0 warnings.
[convert] [ INFO: ] conversion completed with 0 errors and 0 warnings.
[copy] Copying 3 files to C:\JCDK3.0.2_ClassicEdition\samples\classic_applets\Service\applet\build

generate_script:
[scriptgen] Java Card Ant Tasks, Version 1.0
[scriptgen] Copyright (c) 2009 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
[scriptgen] Note: Usage of Java Card Ant Tasks is optional. Java Card Ant Tasks are not supported by Sun Microsystems, Inc.
[scriptgen] ScriptGen [v3.0.2]
[scriptgen] Copyright (c) 2009 Sun Microsystems, Inc.
[scriptgen] All rights reserved.
[scriptgen] Use is subject to license terms.
[scriptgen] APDU script file for CAP file download generated.

run:

all:

BUILD SUCCESSFUL
Total time: 6 seconds
C:\JCDK3.0.2_ClassicEdition\samples\classic_applets\Service\applet>
```

3. Perform any additional actions required by the individual sample's run procedure.

Additional actions might include restarting the RI and using `ant` with an appropriate target to run additional APDU scripts generated by the build. These actions are described in the procedures used to run each sample.

Running the `classic_applets` Samples

The following sections describe the following development kit samples in order of their complexity and provide procedures for running them:

- `HelloWorld` sample - Demonstrates the base structure of a Java Card 3 platform applet that developers can use to develop, deploy, create, execute, delete, and unload a stand-alone module.

`HelloWorld` is a minimal applet utilizing the simplest source code and meta-files. See [“HelloWorld Sample” on page 25](#).

- `Channels` sample - Demonstrates the use of logical channels which allows selecting multiple applets at the same time.

See [“Channels Sample” on page 26](#).

- `Service` sample - Demonstrates the Java Card 3 platform service framework of classes and interfaces that enable a Java Card technology-based applet to be designed as an aggregation of service components.

See [“Service Sample” on page 28](#).

- `Utility` sample - Demonstrates the use of the utility APIs in an applet to simulate stock trading and portfolio management.

See [“Utility Sample” on page 29](#).

- `Wallet` sample - Demonstrates a simplified cash card application.

See [“Wallet Sample” on page 31](#).

- `ObjectDeletion` samples - Contains two samples, `odDemo1` and `odDemo2`, that demonstrate applet and package deletion, as well as the object deletion mechanism which removes unreachable objects.

See [“ObjectDeletion Sample” on page 32](#).

- `PhotoCard` sample - Demonstrates how to store images in the large address space that is available in the 32-bit version of the Java Card 3 Platform, Classic Edition reference implementation.

See [“PhotoCard Sample” on page 36](#).

- RMIPurse sample - Demonstrates the use of the Java Card platform Remote Method Invocation (Java Card RMI) API.

The basic example used is a program that manages a counter remotely, and is able to decrement, increment, and return the value of an account. See [“RMIPurse Sample” on page 38](#) and [Chapter 15, “Programming to the Java Card RMI Client-Side API” on page 139](#).

- SecureRMIPurse sample - Similar to the RMIPurse sample, but demonstrates additional security at the transport level.

This sample is only included in bundles intended solely for distribution inside the U.S. See [“SecureRMIPurse Sample” on page 39](#).

- SignatureMessageRecovery sample - Demonstrates message recovery.

This sample is only included in bundles intended solely for distribution inside the U.S. See [“SignatureMessageRecovery Sample” on page 42](#).

HelloWorld Sample

The HelloWorld sample demonstrates the base structure of a Java Card 3 platform applet that developers can use to develop, deploy, create, execute, delete, and unload a stand-alone module.

▼ Run the HelloWorld Sample

1. Open a Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
- b. Start the RI by entering the following command at the command prompt:

```
cref
```

Note – `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\HelloWorld\applet` directory.

b. Enter the `ant all` command at the command prompt.

In this sample, the **`ant all`** command builds the applet, executes the APDU script, and creates an output file in the `applet` directory. The ant script names the output file either `default.out` or the custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

3. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `HelloWorld.expected.out` file.

Channels Sample

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

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

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

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

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

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

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

When the user wants to use the service, the `AccountAccessor` applet is selected on another logical channel so that the terminal can query the balance. The `AccountAccessor` can return the balance only if the `ConnectionManager` is

active. The `ConnectionManager` applet sets the connection and tracks the connection status. Based on the value of an area code variable, the `ConnectionManager` determines whether the connection is local or remote. It also determines whether the connection is contacted or contactless. `AccountAccessor` uses this information to debit the account at the appropriate rate. The connection is disabled when the user completes the call or when the account is depleted.

▼ Run the Channels Sample

1. Open a Command Prompt window and perform the following:
 - a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
 - b. Start the RI by entering the following command at the command prompt:

```
cref
```

Note – `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\Channels\applet` directory.

- b. Enter the `ant all` command at the command prompt.

In this sample, the **ant all** command builds the applet, executes the APDU script, and creates an output file in the `applet` directory. The ant script names the output file either `default.out` or the custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

3. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `Channels.expected.out` file.

Service Sample

Java Card platform provides a service framework of classes and interfaces that allow a Java Card technology-based applet to be designed as an aggregation of service components. Service demo essentially demonstrates this. The class `Main.java` adds a `TestService` to process the APDUs dispatched by the client. Based on the contents of `INS` command in the APDU sent it does the following:

- If `INS` is `0x10`, it returns status word `6617`.
- If `INS` is `0x20`, it returns status word `6618`.
- If `INS` is `0x30`, it returns status word `9000`.

▼ Run the Service Sample

1. Open a Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
- b. Start the RI by entering the following command at the command prompt:

```
cref
```

Note – `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\Service\applet` directory.
- b. Enter the `ant all` command at the command prompt.

In this sample, the **ant all** command builds the applet, executes the APDU script, and creates an output file in the `applet` directory. The ant script names the output file either `default.out` or the custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the `APDUtool` execution to the *outputfile_name* file.

3. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `service.expected.out` file.

Utility Sample

The `Utility` sample demonstrates how the utility APIs can be used in an application. This applet is a simple version of a hypothetical broker applet that is used to assist the user in buying and selling stocks. The applet uses constructed TLVs and primitive TLVs to manage the portfolio. The communication with the broker is also in the form of TLVs and uses the math API to determine the value of a trade. It also uses the new integer API to construct an integer from byte array and set integers in byte arrays for TLV objects.

This applet provides the following features:

- PIN protected access to the application.
- Storage of portfolio information on the card.
- Retrieval of complete portfolio information from the card.
- Retrieval of information on a particular stock in the portfolio.
- Assistance for the user in creating a stock purchase request for the broker.
- Assistance the user in creating a sell stock request for the broker.
- On receiving a trade confirmation, update the portfolio accordingly.
- Get information on current user account balance.

PIN Protection

Uses the standard PIN API in the Java Card platform to protect access to the applet.

Storage of Portfolio

The applet uses a portfolio constructed TLV to store the information regarding all the stocks that the user currently holds. The information is stored in the form of `stockInfo` constructed TLV. Each `stockInfo` TLV contains the following:

- Stock symbol
- Number of stocks
- Last Trade Constructed TLV
 - Number of stocks
 - Stock Price

Stock Trading

The applet assists the user in buying and selling stocks by creating a “signed” purchasing or selling request for the broker in the form of a stock purchase request constructed TLV or sell stock request constructed TLV. Before the request is generated, the applet checks to see if the user has enough stocks in case the request is to sell the stock and enough account balance if the request is to buy new stock. The request is sent back to the terminal where the terminal application may retrieve the TLV from the response APDU and send it to the broker.

If the trade is successful, the broker sends back a confirmation message in the form a sell confirmation TLV or purchase confirmation TLV. The applet retrieves the information from the confirmation TLV and updates the portfolio as follows:

- If a new stock is bought, the applet creates a new constructed `stockInfo` TLV to store the new stock information.
- If the user already had a stock, the number of stocks the user currently holds, and the last trade information is updated accordingly.
- If as a result of the trade the user has 0 stocks of a certain company, the `stockInfo` TLV for that stock is removed from the portfolio constructed TLV.

Get Information On a Stock

User may use this feature to get information regarding a specific stock rather than retrieving the whole portfolio. If a stock is not found, the appropriate exception is thrown. The information is returned in the form of a `stockInfo` TLV that contains the following:

- Stock symbol
- Number of stocks
- Last trade constructed TLV
- Number of stocks
- Stock price

▼ Run the Utility Sample

1. **Open a Command Prompt window and perform the following:**
 - a. **Navigate to the `JC_CLASSIC_HOME\bin` directory.**
 - b. **Start the RI by entering the following command at the command prompt:**

```
cref
```

Note – `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:

a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\Utility\applet` directory.

b. Enter the `ant all` command at the command prompt.

In this sample, the **`ant all`** command builds the applet, executes the APDU script, and creates an output file in the `applet` directory. The ant script names the output file either `default.out` or the custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

3. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `utility.expected.out` file.

Wallet Sample

The `Wallet` sample demonstrates a simplified cash card application. It keeps a balance, and exercises some of the Java Card API features such as the use of a PIN to control access to the applet.

The script file `wallet.scr` contains the sequence in which this is done.

▼ Run the Wallet Sample

1. Open a Command Prompt window and perform the following:

a. Navigate to the `JC_CLASSIC_HOME\bin` directory.

b. Start the RI by entering the following command at the command prompt:

```
cref
```

Note – `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:

a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\Wallet\applet` directory.

b. Enter the `ant all` command at the command prompt.

In this sample, the **`ant all`** command builds the applet, executes the APDU script, and creates an output file in the applet directory. The ant script names the output file either `default.out` or the custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

3. Verify that the contents of the output file in the applet directory are the same as the contents of the `wallet.expected.out` file.

ObjectDeletion Sample

The sample generates the following four APDU scripts that demonstrate the object deletion mechanism, applet deletion, and package deletion:

- `odDemo1-1.scr` - Demonstrates the object deletion mechanism and verifies that memory for objects referenced from transient memory of type `CLEAR_ON_DESELECT` is reclaimed after an applet is deselected.

`odDemo1-1.scr` does not depend on any other sample. The final state of `cref` memory must be saved to a file for `odDemo1-2.scr` to use.

- `odDemo1-2.scr` - Demonstrates the object deletion mechanism and verifies that memory for objects referenced from transient memory of type `CLEAR_ON_RESET` is reclaimed after card reset.

The `odDemo1-2.scr` sample must be run after `odDemo1-1.scr` because the initial state of `cref` must be the same as its final state after running `odDemo1-1.scr`. After running `odDemo1-2.scr`, the final state of `cref` must be saved to a file so it can be used by `odDemo1-3.scr`.

- `odDemo1-3.scr` - Performs applet deletion, package deletion, and employs the `AppletEvent.uninstall` method to uninstall an applet. The sample verifies that all transient memory of type `CLEAR_ON_RESET` and `CLEAR_ON_DESELECT` is returned to the memory manager. The sample also demonstrates the use of the `AppletEvent.uninstall()` method.

The `odDemo1-3.scr` sample must be run after `odDemo1-2.scr` because the initial state of `cref` must be the same as its final state after running `odDemo1-2.scr`.

- `odDemo2.scr` - Demonstrates package deletion and checks that persistent memory is returned to the memory manager. This sample has one script, `odDemo2.scr`.

The four APDU scripts are run individually from a Command Prompt window. The RI must be restarted before running each APDU script.

▼ Run the ObjectDeletion Sample

1. Open a Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
- b. Start the RI by entering the following command at the command prompt:

```
cref -o e2p
```

2. In a different Command Prompt window, perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\ObjectDeletion\applet` directory.
- b. Enter the `ant all` command at the command prompt.

In this sample, the **ant all** command generates the APDU script.

3. In the `cref` Command Prompt window, stop the RI by using **ctrl + c**.
4. In the `cref` Command Prompt window, restart the RI by entering the following command:

```
cref -o e2p -i e2p
```
5. In the applet Command Prompt window, enter the following command at the command prompt:

```
ant run1-1
```

The **ant run1-1** command executes the `od1-1.scr` APDU script and creates an output file in the `applet` directory. The ant script names the output file either `default.out` or the custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

6. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `od1-1.expected.out` file.

7. In the `cref` Command Prompt window, restart the RI by entering the following command:

```
cref -o e2p -i e2p
```

8. In the `applet` Command Prompt window, enter the following command at the command prompt:

```
ant run1-2
```

The **ant run1-2** command executes the `od1-2.scr` APDU script and creates an output file (`default.out`) in the `applet` directory. See [Step 5](#) for the command line required to specify a custom output file name.

9. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `od1-2.expected.out` file.

10. In the `cref` Command Prompt window, restart the RI by entering the following command:

```
cref -o e2p -i e2p
```

11. In the `applet` Command Prompt window, enter the following command at the command prompt:

```
ant run1-3
```

The **ant run1-3** command executes the `od1-3.scr` APDU script and creates an output file (`default.out`) in the `applet` directory. See [Step 5](#) for the command line required to specify a custom output file name.

12. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `od1-3.expected.out` file.

13. In the `cref` Command Prompt window, restart the RI by entering the following command:

```
cref -o e2p -i e2p
```

14. In the applet Command Prompt window, enter the following command at the command prompt:

```
ant run2
```

The **ant run2** command executes the `od2.scr` APDU script and creates an output file (`default.out`) in the applet directory. See [Step 5](#) for the command line required to specify a custom output file name.

15. Verify that the contents of the output file in the applet directory are the same as the contents of the `od2.expected.out` file.

16. In the `cref` Command Prompt window, restart the RI by entering the following command:

```
cref -o e2p -i e2p
```

17. In the applet Command Prompt window, enter the following command at the command prompt:

```
ant run2-2
```

The **ant run2-2** command executes the `od2-2.scr` APDU script and creates an output file (`default.out`) in the applet directory. See [Step 5](#) for the command line required to specify a custom output file name.

18. Verify that the contents of the output file in the applet directory are the same as the contents of the `od2-2.expected.out` file.

19. In the `cref` Command Prompt window, restart the RI by entering the following command:

```
cref -o e2p -i e2p
```

20. In the applet Command Prompt window, enter the following command at the command prompt:

```
ant run3
```

The **ant run3** command executes the `od3.scr` APDU script and creates an output file (`default.out`) in the applet directory. See [Step 5](#) for the command line required to specify a custom output file name.

21. Verify that the contents of the output file in the applet directory are the same as the contents of the `od3.expected.out` file.

22. In the `cref` Command Prompt window, restart the RI by entering the following command:

```
cref -o e2p -i e2p
```

23. In the applet Command Prompt window, enter the following command at the command prompt:

```
ant run3-2
```

The **ant run3-2** command executes the `od3-2.scr` APDU script and creates an output file (`default.out`) in the applet directory. See [Step 5](#) for the command line required to specify a custom output file name.

24. **Verify that the contents of the output file in the applet directory are the same as the contents of the `od3-2.expected.out` file.**

PhotoCard Sample

The PhotoCard sample illustrates how to use the large address space available in the 32-bit version of the RI. The sample uses the large address space of the smart card's EEPROM memory to store up to four GIF images. The images are included with the sample.

The PhotoCard sample consists of two parts: a card applet and a client program communicating with it. The `photocard` sample employs a collection of arrays to store large amounts of data. The arrays allow the applet to take advantage of the platform's capabilities by transparently storing data.

The design and coding of applications that use the large address space to access memory must adhere to the target platform's requirements. Smart cards have limited resources, code cannot be guaranteed to behave identically on different cards. For example, if the `photocard` applet runs on a card with less mutable persistent memory available for storage, it might run out of memory space when it attempts to store the images. A set of inputs might not produce the same set of outputs in an RI with different characteristics. The applet code must account for any different implementation-specific behavior.

▼ Run the PhotoCard Sample

1. **Open a Command Prompt window and perform the following:**

- a. **Navigate to the `JC_CLASSIC_HOME\bin` directory.**
- b. **Start the RI by using the following command at the command prompt:**

```
cref -o demoe
```

Starting the RI with the `-o demoe` option and filename causes the RI to save the EEPROM contents to a file named `demoe`. See [Chapter 9](#) for more information about using `cref` and its command line options.

2. **Open a second Command Prompt window and perform the following:**

a. **Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\PhotoCard\applet` directory.**

b. **Enter the following command at the command prompt:**

```
ant all
```

In this sample's applet directory, the **ant all** command executes the APDU script, installs the photocard application, and creates an output file (`default.out`) in the applet directory.

3. **Verify that the contents of the output file in the applet directory are the same as the contents of the `photocard-applet.expected.out` file.**

4. **In the `cref` Command Prompt window, restart the RI by using the following command:**

```
cref -z -i demoe
```

Starting the RI with the `-z -i demoe` option and filename causes the RI to use the contents of the `demoe` file to initialize the EEPROM and to display the resource consumption statistics. See [Chapter 9](#) for more information about using `cref` and its command line options.

5. **In the applet Command Prompt window, perform the following:**

a. **Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\PhotoCard\client` directory.**

b. **Enter the following command at the command prompt:**

```
ant all
```

In this sample's client directory, the **ant all** command executes the APDU script and an output file (`actual_output.txt`) in the applet directory.

6. **Verify that the contents of the `actual_output.txt` file are the same as the contents of the `photocard-client.expected.out` file.**

Note – Photo verification requires availability of MessageDigest class and SHA256 algorithm. If these are not available, the `actual_output.txt` file will not contain the last line of the `photocard-client.expected.out` file (Photo is valid).

RMIPurse Sample

A Java Card RMI application consists of two parts: a card applet and a client program communicating with it. In this case, the RMIPurse applet is installed in EEPROM image. For further details see [Chapter 15, “Programming to the Java Card RMI Client-Side API” on page 139](#).

The RMIPurse sample uses the card applet `PurseApplet`, the `Purse` interface and its implementation `PurseImpl`. These classes reside in the package `com.sun.javacard.samples.RMIDemo`. The client-side program `PurseClient` resides in the package `com.sun.javacard.clientsamples.purseclient`.

The `Purse` interface describes the supported functionality: methods for obtaining the account balance, debiting and crediting the account, and obtaining and setting an account number. The interface also defines the constants used for error reporting. The `PurseImpl` class implements `Purse`.

The card applet, `PurseApplet`, creates and registers instances of the dispatcher and the Java Card RMI service.

The client-side program, `PurseClient`, represents a simple Java Card RMI client. The program opens a connection with a card, creates the Java Card RMI Connect instance, and selects the Java Card applet (in this case, the `PurseApplet`). The program then gets the initial reference from `PurseApplet` (the reference to an instance of `PurseImpl`) and casts it to the `Purse` interface type. This allows `PurseImpl` to be treated as a local object. The program can then exercise the card by debiting and crediting different amounts, and by setting and getting the account number. The program demonstrates error handling by intentionally attempting to set an account number of incorrect size. This causes a `UserException` to be thrown with the appropriate error code.

The client part of the `RMIDemo` can be run without parameters or with the `-i` parameter:

- If the sample is run without parameters, remote references are identified using the class name of the remote object.
- If the sample is run with the `-i` parameter, remote references are identified using the list of remote interfaces implemented by the remote object.

For more information on these formats, see Chapter 8 of the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*.

▼ Run RMIPurse

1. Open a Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\bin` directory.

- b. Start the RI by using the following command at the command prompt:

```
cref -o demoe
```

Starting the RI with the `-o demoe` option and filename causes the RI to save the EEPROM contents to a file named `demoe`. See [Chapter 9](#) for more information about using `cref` and its command line options.

2. Open a second Command Prompt window and perform the following:

- a. Navigate to the

`JC_CLASSIC_HOME\samples\classic_applets\RMPurse\applet` directory.

- b. Enter the following command at the command prompt:

```
ant all
```

In this sample's `applet` directory, the **ant all** command executes the APDU script and installs the RMI application.

3. In the `cref` Command Prompt window, restart the RI by using the following command:

```
cref -i demoe
```

Starting the RI with the `-i demoe` option and filename causes the RI to use the contents of the `demoe` file to initialize the EEPROM. See [Chapter 9](#) for more information about using `cref` and its command line options.

4. In the `applet` Command Prompt window, perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\RMPurse\client` directory.

- b. Enter the following command at the command prompt:

```
ant all
```

In this sample's `client` directory, the **ant all** command executes the APDU script and generates the `rmidemo.actual.output` file.

5. Verify that the contents of the `rmidemo.actual.output` file in the `client` directory are the same as the contents of the `rmidemo.expected.output` file in the `RMPurse` directory.

SecureRMPurse Sample

This sample is available only in bundles intended solely for distribution inside the U.S.

The SecureRMIPurse sample is a version of RMIPurse with an added security service. SecureRMIPurse uses the card applet SecurePurseApplet, the Purse interface and its implementation SecurePurseImpl, and a definition of the security service MySecurityService. These classes reside in the package `com.sun.javacard.samples.SecureRMIDemo`. The sample also uses the client-side program SecurePurseClient and the specialized card accessor CustomCardAccessor. These classes reside in the package `com.sun.javacard.clientsamples.SecurePurseClient`.

The Purse interface is similar to the interface used in the non-secure case, however, there is an extra constant: `REQUEST_DENIED`. This constant is used to report situations where the client tries to invoke a method that it is not allowed to access.

The MySecurityService class is a security service that is responsible for ensuring data integrity by verifying checksums on incoming commands and attaching checksums to outgoing commands. The program also requires the client to authenticate itself as the principal application provider or principal cardholder by sending a two-byte PIN.

The implementation of Purse, SecurePurseImpl, is similar to the non-secure case, however, at the beginning of each method call, a call is made to the security service that ensures that the business rules are satisfied and that the data is not corrupted.

The applet, SecurePurseApplet, is similar to the non-secure case, with the exception that it creates and registers an instance of MySecurityService.

The client-side program, SecurePurseClient, is similar to the non-secure case, with the exception that instead of a generic card accessor, it uses its own implementation, CustomCardAccessor, to perform additional preprocessing and postprocessing of data and to support the additional command, `authenticateUser`.

SecurePurseClient also requires verification of the user. After the applet is inserted, a PIN must be given to the card-side applet by calling `authenticateUser` on CustomCardAccessor.

When `authenticateUser` is called, CustomCardAccessor prepares and sends the following command:

TABLE 4-3 Authenticate User Command

CLA_AUTH	INS_AUTH	P1 field	P2 field	LC field	PIN (two bytes)	
0x80	0x39	0	0	2	xx	xx

On the card side, MySecurityService processes the command. If the PIN is correct, then the appropriate flags are set in the security service and a confirmation response is returned to the client. Once authentication is passed, the client program receives the balance, credits the account, and again receives the balance. The

program demonstrates error handling when the client attempts to debit a number of units from the account. This causes the program to throw a `UserException` with the code `REQUEST_DENIED`.

As with `RMIDemo`, the client part of the `SecureRMIDemo` can be run without parameters or with the `-i` parameter:

- If the sample is run without parameters, remote references are identified using the class name of the remote object.
- If the sample is run with the `-i` parameter, remote references are identified using the list of remote interfaces implemented by the remote object.

For more information on these formats, see Chapter 8 of the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*.

▼ Run SecureRMIPurse

1. Open a Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
- b. Start the RI by using the following command at the command prompt:

```
cref -o demoe
```

Starting the RI with the `-o demoe` option and filename causes the RI to save the EEPROM contents to a file named `demoe`. See [Chapter 9](#) for more information about using `cref` and its command line options.

2. Open a second Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\SecureRMIPurse\applet` directory.
- b. Enter the following command at the command prompt:

```
ant all
```

In this sample's applet directory, the **ant all** command executes the APDU script and installs the secure RMI application.

3. In the `cref` Command Prompt window, restart the RI by using the following command:

```
cref -i demoe
```

Starting the RI with the `-i demoe` option and filename causes the RI to use the contents of the `demoe` file to initialize the EEPROM. See [Chapter 9](#) for more information about using `cref` and its command line options.

4. In the applet Command Prompt window, perform the following:

a. **Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\SecureRMIPurse\client` directory.**

b. **Enter the following command at the command prompt:**

```
ant all
```

In this sample's `client` directory, the **ant all** command executes the APDU script that generates the `securermidemo.expected.out` file.

5. **Verify that the contents of the `securermidemo.expected.output` file in the client directory are the same as the contents of the `securermidemo.expected.output` file in the `SecureRMIPurse` directory.**

SignatureMessageRecovery Sample

This sample is available only in bundles intended solely for distribution inside the U.S.

Message recovery refers to the mechanism whereby part of the message used to create the message digest is also included as padding in the signature block. During signature verification, the message data padding does not need to be explicitly sent to the verifying entity, it can automatically be extracted from the signature block.

Message Recovery Order of Operations

This section describes the order of operations for signing and verifying.

Signing

1. The user invokes a combination of the `update` and `sign` methods to generate a signature based on message data provided by the user.
2. The `sign` method returns an indication to the user of the portion of the message that was included as padding in the signature.

This is required so that the user knows what remaining data must still be sent along with the signature block.

Verifying

1. The user initializes the signature object with signature at the very beginning so it can get the recoverable data at the earliest.

2. The user invokes a combination of the update and verify methods to verify the signature based on the message data provided by the user.
3. The verify method verifies the signature by comparing the accumulated hash with the hash in the message representative recovered during initialization.

Sample Application

This sample consists of two scripts representing two scenarios for Signature with Message Recovery. The first script, `sigMsgFullRec.scr`, shows the scenario in which the message to sign is small enough that the entire message itself becomes part of the signature padding (hence the name “Full Recovery” since you can recover the full message from the signature itself). The second script, `sigMsgPartRec.scr`, demonstrates the scenario in which the message to sign is large enough that only some part of it is included in the signature padding (hence the name “Partial Recovery”). The scenarios are detailed next:

sigMsgFullRec.scr Script

The sequence of events resulting from running this script are:

1. The script sends to the sample application a small message to sign.
2. The application initializes the signature object with the algorithm `Signature.ALG_RSA_SHA_ISO9796_MR` and signs the message. Because the message is small enough, the application returns the signature data to the script.
3. The script then simulates the verification phase in which it sends the signature data to the sample application asking it to verify the message.

The application recovers the original message from the signature data and also verifies the signature, then returns the original data back to the script. (If the signature verification fails, it returns an error code).

sigMsgPartRec.scr Script

The sequence of events resulting from running this script are:

1. The script sends to the sample application a large message to be signed.
2. The application initializes the signature object with algorithm `Signature.ALG_RSA_SHA_ISO9796_MR` and signs the message. Because the message is too large to fit in the signature, the application returns back to the script the number of bytes of original message that is embedded in the signature data. The application also returns back to the script the signature data.

3. The script then simulates the verification phase in which it sends the signature data to the sample application.
4. The application recovers the partial message and returns back to the script.
5. The script sends the remainder of the message to the application to verify the signature.
6. The application verifies the signature against the entire message and returns success.

▼ Run SignatureMessageRecovery

1. Open a Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
- b. Start the RI by entering the following command at the command prompt:

```
cref
```

Note – `cref` command options are not required in this sample.

2. In a different Command Prompt window, perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\classic_applets\SignatureMessageRecovery\applet` directory.
- b. Enter the following command at the command prompt:

```
ant run1
```

The **ant run1** command builds the applet and runs the `sigMsgPartRec.scr` script that generates the `sigMsgPartRec.actual.output` file.

3. Verify the contents of the `sigMsgPartRec.actual.output` file in the applet directory are the same as the contents of the `sigMsgPartRec.expected.output` file in the `SignatureMessageRecovery` directory.
4. In the `cref` Command Prompt window, restart the RI by using the following command:

```
cref
```
5. In the applet Command Prompt window, enter the following command at the command prompt:

```
ant run2
```

The **ant run2** command builds the applet and runs the `sigMsgFullRec.scr` script that generates the `sigMsgfullRec.actual.output` file.

6. **Verify the contents of the `sigMsgfullRec.actual.output` file are the same as the contents of the `sigMsgfullRec.expected.output` file.**

Running the reference_apps Samples

The following sections describe the reference applet demonstrations and how to run them:

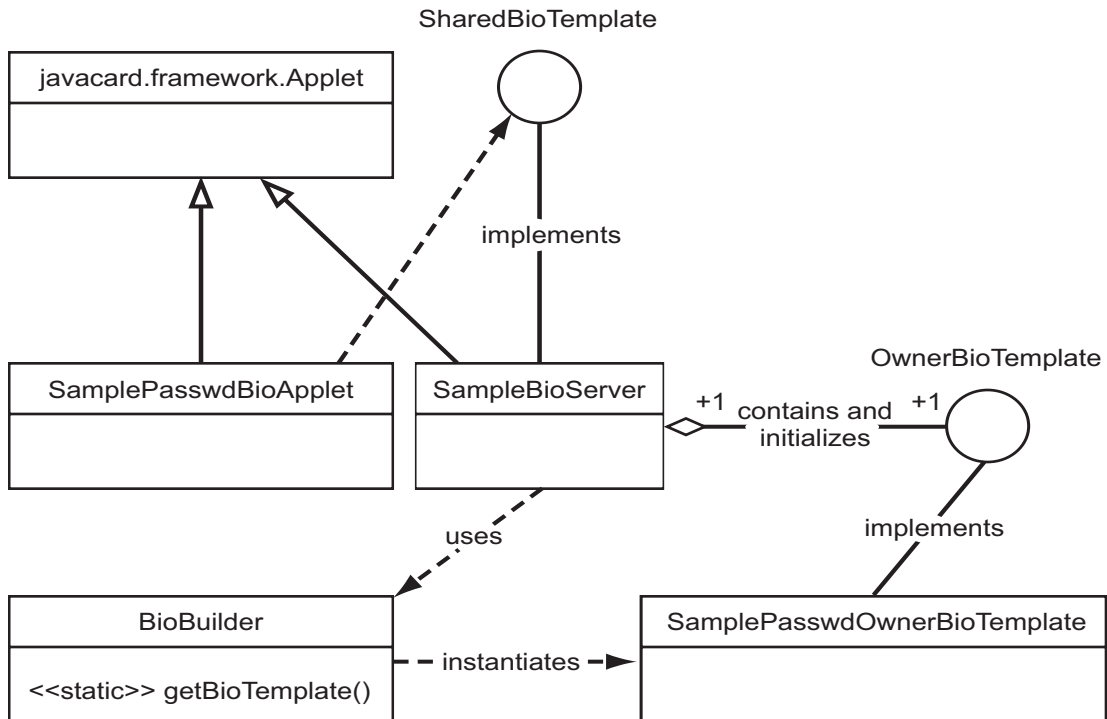
- **Biometry Sample Application** - Demonstrates the use of the biometric APIs of type PASSWORD.
See [“Biometry Sample Application” on page 45](#).
- **JavaPurseCrypto Sample** - Demonstrates the use of a DES MAC algorithm.
This sample is only included in bundles intended solely for distribution inside the U.S. See [“JavaPurseCrypto Sample” on page 52](#).
- **PurseWithLoyalty Sample Application** - Demonstrate the use of shareable interfaces.
See [“JavaPurse Sample Application” on page 50](#).
- **Transit Sample** - Demonstrates a contactless card-based transit applet and its interaction with a turnstile transit terminal and with a point of sale terminal.
This sample is only included in bundles intended solely for distribution inside the U.S. See [“Transit Sample” on page 53](#).

Biometry Sample Application

In this sample, a user password is enrolled on the card and a candidate password is matched against the enrolled password demonstrating the basic functionality of the biometric API. See [“How the Biometric API Works” on page 47](#).

The steps described in [FIGURE 4-5](#) illustrate the sequence of events that occur when the sample application uses the biometric API. Note that the sequence of steps depicted is a scenario in which everything works well. In other usages, other sequences of steps occur when things do not work well (such as an error occurring during the enrollment process, the matching process, the card-blocked state, or the non-initialized state). [FIGURE 4-5](#) also illustrates the sequence for enrolling the PASSWORD bio-template done by `SampleBioServer`.

FIGURE 4-5 Biometric Sample Sequence Diagram



1. The off-card tool (see [“Off-card Tool” on page 47](#)) takes a hard coded password and sends it to the card for enrollment.

The applet selected on-card is the `SampleBioServer` applet. See [“SamplePasswdBioServer Class” on page 47](#).

2. The `SampleBioServer` applet stores the password as the reference template with a hard coded number of tries allowed before block (5).
3. For matching, the APDUscript asks the on-card client (`SamplePasswdBioApplet`) to ask the `SharedBioTemplate` for the public template.

For this sample, the public template only contains the version number of the implementation and the length of stored password representing the requirement for password capture. See [“SamplePasswdBioApplet Class” on page 47](#) and [“SamplePasswdOwnerBioTemplate Class” on page 47](#).

4. The script sends for matching the same password used for enrollment.

The card has a matching algorithm and calculates the score based on the stored password and received password.

5. The card returns verification successful to the script.

Off-card Tool

For this sample, the off-card tool is a simple `apdutool` script used for both enrolling and matching.

SamplePasswdOwnerBioTemplate Class

This class implements the `OwnerBioTemplate` interface and is what the `BioBuilder` constructs when asked for a `OwnerBioTemplate` interface for the `BioBuilder.PASSWORD` bio-type. This class provides the enrollment and matching capability to clients.

SamplePasswdBioServer Class

This class represents the `BioServer` applet on the card. It is responsible for communicating with off-card clients with APDUs and with on-card client applets with an implementation of `ShareableBioTemplate` that it implements. This class causes the enrolling of the password biometric while communicating with an off-card tool that sends the password down to the `BioServer`. This class is also the interface to the on-card and off-card clients for the biometric functionality on the card.

SamplePasswdBioApplet Class

This represents an on-card client applet for the password biometric sample. It communicates with an off-card tool to get the password and calls the `match` method on the `ShareableBioTemplate` reference it gets from the Java Card runtime environment, which is given the `SamplePasswdBioServer` applet AID.

How the Biometric API Works

The biometric API is designed to perform three basic functions:

- Match biometric information on-card
- Enroll users off-card and transfer their information on-card
- Verify the user in a sequence of off-card and on-card interactions

On-card Matching

One of the requirements of the architecture is that biometric verification must happen on-card for security reasons. The card cannot send out a person's biometric information for verification to be done off-card. The reasoning here is the same as for a PIN, which is that it would not be secure to do so.

Enrollment Process

During the enrollment process, a person's biometric information is captured off-card and then transferred on-card for storage and verification purpose. Since Java Card technology-based cards are generally limited in their resources, the entire data captured off-card is not sent to the card. What is sent is a digested version of the biometric data and is very specific to a particular algorithm. For this sample, however, a password is small enough that the entire password is transferred to the card.

The user-specific data transferred makes up a reference template that is used later for verification. At the end of the enrollment process, there also exists an associated public template. The public template consists of information for the off-card tool to capture the relevant information from the user during verification.

For example, in the Precise Biometrics implementation of the fingerprint biometric API, the public template contains the coordinates, relative to the reference point for capturing fingerprint information. The off-card tool looks at these coordinates and extracts that information from the user. The public template defines the data requirements for verification. For this sample, the public template does not contain any such specification since the entire password is compared. In the sample, the public template just contains version information.

Verification Process

During the verification process the user enters biometric information into a sensor or input device. The information gathered from the user input is defined by the public template (see [“Enrollment Process” on page 48](#)). This information might be pre-processed off-card and transferred to the card for verification. The on-card biometric application performs the verification given the reference template with pre-existing user information and the new information that came in. The following describe the verification sequence:

1. The host issues a verification request to the card.
2. The card returns the public template to the host.
3. The host captures user information and extracts the data defined by the public template.

The host might perform data-processing specific to the biometric algorithm.

4. The host sends extracted verification data to the card.
5. The card matches the captured data with its own representation stored in the reference template.

The matching process results in a score of how well the user information matches the reference template information.

6. The card compares the score with the threshold for acceptable criteria and returns the verification result to the host.

Implementation Notes

The following restrictions apply for the Sun Microsystems implementation of the password biometric:

- The minimum password length to be enrolled must be 5 bytes.
- The maximum password length to be enrolled must be 50 bytes.

The array containing password data during enrollment or matching must have the password laid out as a byte array with each character represented by a byte starting from index `offset`. There can be no other information in the byte array from index `offset` to index `offset+length-1`. For example, password “tests” must be represented by the byte array {116, 101, 115, 116, 115} starting at index 0 with length 5.

The public template for the stored password returned during a matching session is a byte array (`dest`) with formatting as shown below. The version for this implementation is 1.0.0, so the `dest` array would be as follows, where *passwd length* represents the length of the enrolled password.

- `dest[0]=1`
- `dest[1]=0`
- `dest[2]=0`
- `dest[3]=passwd length`

▼ Run the Biometry Sample

1. Open a Command Prompt window and perform the following:
 - a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
 - b. Start the RI by using the following command at the command prompt:

```
cref -o e2p
```

Starting the RI with the `-o e2p` option and filename causes the RI to save the EEPROM contents to a file named `e2p`. See [Chapter 9](#) for more information about using `cref` and its command line options.

2. Open a second Command Prompt window and perform the following:

a. Navigate to the `JC_CLASSIC_HOME\samples\reference_apps\Biometry\Server\applet` directory.

b. Enter the following command at the command prompt:

```
ant all
```

In this sample's applet directory, the **ant all** command executes the APDU script and installs the secure RMI application.

3. In the `cref` Command Prompt window, restart the RI by using the following command:

```
cref -i e2p
```

Starting the RI with the `-i e2p` option and filename causes the RI to use the contents of the `e2p` file to initialize the EEPROM. See [Chapter 9](#) for more information about using `cref` and its command line options.

4. In the applet Command Prompt window, perform the following:

a. Navigate to the `JC_CLASSIC_HOME\samples\reference_apps\Biometry\Client\applet` directory.

b. Enter the following command at the command prompt:

```
ant all
```

In this sample's client directory, the **ant all** command executes the APDU script.

5. Verify that the output displayed in the Command Prompt window is the same as the contents of the `biometry-client.expected.out` file.

JavaPurse Sample Application

The JavaPurse sample application consists of two components, a JavaPurse applet and a JavaLoyalty applet.

The JavaPurse applet demonstrates a simple electronic cash application. The applet is selected and initialized with various parameters such as the Purse ID, the expiration date of the card, the Master and User PINs, maximum balance, and maximum transaction. Transaction operations perform the actual debits and credits

to the electronic purse. If a configured loyalty applet is assigned for the CAD performing the transaction, `JavaPurse` communicates with it to grant loyalty points. In this sample, `JavaLoyalty` is the provided loyalty applet.

A number of transaction sessions are simulated where amounts are credited and debited from the card. In an additional session, transactions with intentional errors are attempted to demonstrate the security features of the card.

The `JavaLoyalty` applet is a minimalistic loyalty applet that interacts with the `JavaPurse` applet and demonstrates the use of shareable interfaces. The shareable `JavaLoyaltyInterface` is defined in a separate library package, `com.sun.javacard.SampleLibrary`.

`JavaLoyalty` applet is registered with `JavaPurse` when a Parameter Update APDU command with an appropriate parameter tag is executed, and when the AID part of the parameter corresponds to the AID of the `JavaLoyalty` applet. The applet contains a `grantPoints` method. This method implements the main interaction with the client. The `grantPoints` method implementing the `JavaLoyaltyInterface` is requested when the first two bytes of the CAD ID in a request by a `JavaPurse` transaction correspond to the two bytes of CAD ID in the corresponding Parameter Update APDU command.

`JavaLoyalty` maintains the balance of loyalty points. The `JavaLoyalty` applet contains methods to credit and debit the account of points and to get and set the balance.

▼ Run the JavaPurse Sample

1. **Open a Command Prompt window and perform the following:**
 - a. **Navigate to the `JC_CLASSIC_HOME\bin` directory.**
 - b. **Start the RI by using the following command at the command prompt:**

```
cref
```
2. **Open a second Command Prompt window and perform the following:**
 - a. **Navigate to the `JC_CLASSIC_HOME\samples\reference_apps\PurseWithLoyalty\JavaPurse\applet` directory.**
 - b. **Enter the following command at the command prompt:**

```
ant all
```

In this sample's applet directory, the **ant all** command executes the APDU script and generates the output file. The ant script names the output file either `default.out` or a custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

3. **Verify that the contents of the output file in the applet directory are the same as the contents of the `javapurse.expected.ouput` file.**

JavaPurseCrypto Sample

This sample is available only in bundles intended solely for distribution inside the U.S.

The JavaPurseCrypto sample application consists of two components, a JavaPurseCrypto applet and a JavaLoyalty applet. The JavaPurseCrypto applet employs a version of JavaPurse that uses a DES MAC algorithm. A DES MAC is a cryptographic signature that uses DES encryption on all or part of a message (APDU). JavaPurseCrypto uses the DES MAC to verify several of the APDUs. Instead of zeros in the signature currently in JavaPurse, it contains a real signature that can be programmatically signed and verified. Other programs that might interact with JavaPurseCrypto are not affected because all signing and verifying of the signature occurs only within JavaPurseCrypto.

The JavaPurseCrypto sample uses transient DES keys. The use of transient DES keys by the sample highlights the fact that the DES cryptography API has been enhanced to eliminate persistent memory usage when transient DES keys are provided. Eliminating the use of persistent memory when transient DES keys are used provides better performance in a contactless applet.

As in the JavaPurse sample, the JavaLoyalty applet is a minimalistic loyalty applet that interacts with JavaPurseCrypto and demonstrates the use of shareable interfaces. See [“JavaPurse Sample Application” on page 50](#) for additional information about the JavaLoyalty applet.

▼ Run the JavaPurseCrypto Sample

1. **Open a Command Prompt window and perform the following:**

a. Navigate to the `JC_CLASSIC_HOME\bin` directory.

b. Start the RI by using the following command at the command prompt:

```
cref
```

2. Open a second Command Prompt window and perform the following:

a. Navigate to the `JC_CLASSIC_HOME\samples\reference_apps\PurseWithLoyalty\JavaPurseCrypto\applet` directory.

b. Enter the following command at the command prompt:

```
ant all
```

In this sample's applet directory, the **ant all** command executes the APDU script and generates the output file. The ant script names the output file either `default.out` or a custom name specified in the command line. To specify a custom name for the output file, use the following command:

```
ant -Dredirect.output=outputfile_name target
```

In this command, *outputfile_name* represents the name of the output file and *target* represents either the `all` or `run` options of the ant command. In this case, the `all` target is used. This command redirects the output from the APDUtool execution to the *outputfile_name* file.

3. Verify that the contents of the output file in the applet directory are the same as the contents of the `javapursecrypto.expected.out` file.

Transit Sample

This sample is available only in bundles intended solely for distribution inside the U.S, and so it can not be used in global bundles.

The Transit sample illustrates a contactless card-based transit applet. This sample consists of the transit applet and two client applications, the `POSTerminal` client application and the `TransitTerminal` client application.

A typical transit scenario is pre-scripted in the `TransitDemo` file, including crediting and checking the balance (a \$99 initial balance) on the transit card at the POS terminal, entering and exiting the transit system through the Turnstile Transit terminal (a \$10 fee for the trip), and finally checking the new balance (an \$89 balance) on the transit card at the POS terminal.

Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the actual output files generated by running this sample will vary from that of the expected output files for the following instructions:

■ CLA:80 INS:30

▼ Run the Transit Sample

The `TransitDemo` or `TransitDemo.bat` script automatically starts and stops `cref` when needed to simulate interaction sessions with the POS terminal and the turnstile transit terminal.

1. Open a Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\bin` directory.
- b. Start the RI by using the following command at the command prompt:

```
cref -o transitCard
```

2. Open a second Command Prompt window and perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\reference_apps\Transit\Transit\applet` directory.
- b. Enter the following command at the command prompt:

```
ant all
```

In this sample's applet directory, the **ant all** command generates the APDU script and downloads the CAP file.

3. In the `cref` Command Prompt window, restart the RI by using the following command:

```
cref -i transitCard -o transitCard
```

Starting the RI with the `-i transitCard -o transitCard` options and filenames causes the RI to use the contents of the `transitCard` file to initialize the EEPROM and to save the EEPROM contents to a file named `transitCard`. See [Chapter 9](#) for more information about using `cref` and its command line options.

4. In the applet Command Prompt window, perform the following:

- a. Navigate to the `JC_CLASSIC_HOME\samples\reference_apps\Transit\Transit\client` directory.
- b. Enter the following command at the command prompt:

```
ant run1
```

In this sample's client directory, the **ant run1** command compiles and builds the `client.jar` and generates the `actual_output1.txt` file.

5. **Verify that the contents of the `actual_output1.txt` file are the same as the contents of the `TransitClient_1.expected.output` file.**

Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output1.txt` file will vary from **the** `TransitClient_1.expected.output` **file** for the following instructions:

- CLA:80 INS:30
- CLA:80 INS:40

6. **In the `cref` Command Prompt window, restart the RI by using the following command:**

```
cref -i transitCard -o transitCard
```

7. **In the applet Command Prompt window, enter the following command at the command prompt:**

```
ant run2
```

In this sample's `client` directory, the **ant** `run2` command compiles and builds the `client.jar` and generates the `actual_output2.txt` file.

8. **Verify that the contents of the `actual_output2.txt` file are the same as the contents of the `TransitClient_2.expected.output` file.**

Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output2.txt` file will vary from **the** `TransitClient_2.expected.output` **file** for the following instructions:

- CLA:80 INS:30
- CLA:80 INS:40

9. **In the `cref` Command Prompt window, restart the RI by using the following command:**

```
cref -i transitCard -o transitCard
```

10. **In the applet Command Prompt window, enter the following command at the command prompt:**

```
ant run3
```

In this sample's `client` directory, the **ant** `run3` command compiles and builds the `client.jar` and generates the `actual_output3.txt` file.

11. **Verify that the contents of the `actual_output3.txt` file are the same as the contents of the `TransitClient_3.expected.output` file.**

Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output3.txt` file will vary from the `TransitClient_3.expected.output` file for the following instructions:

- CLA:80 INS:30
- CLA:80 INS:40

12. **In the `cref` Command Prompt window, restart the RI by using the following command:**

```
cref -i transitCard -o transitCard
```

13. **In the applet Command Prompt window, enter the following command at the command prompt:**

```
ant run4
```

In this sample's `client` directory, the **ant** `run4` command compiles and builds the `client.jar` and generates the `actual_output4.txt` file.

14. **Verify that the contents of the `actual_output4.txt` file are the same as the contents of the `TransitClient_4.expected.output` file.**

Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output4.txt` file will vary from the `TransitClient_4.expected.output` file for the following instructions:

- CLA:80 INS:30
- CLA:80 INS:40

Converting and Exporting Java Class Files

This chapter describes how to use the Converter tool, including the input files it can process and the output it produces. How to work with export files is also described.

The Converter takes as input the class files that make up a Java programming language package. The Converter verifies that class files comply to limitations described in Section 2.2, “Java Card Platform Language Subset” in the *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*.

The Converter can output a CAP file, a Java Card Assembly file, or an export file. The CAP file is a JAR-format file which contains the executable binary representation of the classes in a Java package. 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, Classic Edition*. 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 manifest file and its contents, see [Chapter 7](#).

For more information on the Java Card Assembly file, see [Appendix A](#) and [Chapter 10](#). For more information on export files, see “[Using Export Files](#)” on [page 65](#).

You are responsible for the consistency of your input data. This means that:

- all input class files are compatible with each other.
- export files of imported packages are consistent with class files that were used for compiling the converting package.

The Converter generates the following output files:

- A CAP file for the Java Card 3 Platform, Classic Edition
- A Java Card Assembly file
- An export file

If the package to be converted contains remote classes or interfaces or if the `-debug` option is specified, the Converter generates a CAP file suitable for version 2.2 or greater of the Java Card platform. Otherwise, the Converter generates files that can also be used by version 2.1 of the Java Card platform. To create a CAP file compatible with version 2.1 of the Java Card platform, you must also use export files for Java Card API packages from the Java Card development kit 2.1.x.

If you are converting more than one package with interdependencies, convert the packages in two passes. First, generate only the export files, then, after that, convert the required CAP or Java Card Assembly files. The Converter tool cannot convert more than one package at a time.

If you have a source release, you may choose to convert packages that import other packages. If you are creating Java Card Assembly files to generate a mask, then the major and minor version number of the imported packages must agree with the version number of the package that imports them. Please see [“Version Numbers for Processed Packages” on page 117](#) for more information.

If you have a source release, you can localize locale-specific data associated with the Converter. For more information, see [Chapter 14](#).

Setting Java Compiler Options

Before you use the Converter tool, be sure to compile your Java code properly.

For the most efficient conversion, compile your class files with the Java SDK compiler's `-g` command line option. The `-g` option causes the compiler to generate the `LocalVariableTable` attribute in the class file. The Converter uses this attribute to determine local variable types. If you do not use the `-g` option, the Converter attempts to determine the variable types on its own. This is expensive in terms of processing and might not produce the most efficient code. You must also compile your class files with the `-g` option if you want to generate a debug component in the CAP file by using the Converter's `-debug` option.

Do not compile with the `-O` option. The `-O` option is not recommended on the Java compiler command line, for these reasons:

- this option is intended to optimize execution speed rather than minimize memory usage. Minimizing memory usage is much more important in the Java Card environment.
- the `LocalVariableTable` attribute will not be generated.

Running the Converter

You invoke the Converter at the command line as follows (see [TABLE 5-4](#) for a description of the arguments):

```
converter.bat [options] package-name package-aid major-version .minor-version
```

Note – The file to invoke the Converter is a batch file (`converter.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

The Converter command line options described in detail in [TABLE 5-4](#) and allow you 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 the default.).

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

- **Loads the package** - If the `exportmap` option is set, the converter loads the package from the export path (see [“Specifying an Export Map”](#) on page 66). Loads the class files of the Java package and creates a data structure to represent the package.
- **Subset checking** - Checks for unsupported Java features in class files.

- **Conversion** - Checks for consistency between the applet AIDs and the imported package AIDs.
- **Reference Checking** - Checks that all references are valid, internal referenced items are defined in the package, import items are declared in the export files (see [“Using Export Files” on page 65](#)).

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 JCVM specification.

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

Before writing the export and JCA files, the Converter determines the output file path. The Converter assumes the output files are written into the directory: `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.

TABLE 5-4 Converter Command Line Arguments

Option	Description
<code>-help</code>	Prints help message.
<code>package-name</code>	Fully-qualified name of the package to convert.
<code>package-aid</code>	5- to 16-decimal, hex or octal numbers separated by colons. Each of the numbers must be byte-length.
<code>major-version</code> <code>minor-version</code>	User-defined version of the package.
<code>-applet AID class_name</code>	Sets the default applet AID and the name of the class that defines the applet. If the package contains multiple applet classes, this option must be specified for each class.
<code>-classdir</code> <code>root-directory-of-class hierarchy</code>	Sets the root directory where the Converter will look for classes. If this option is not specified, the Converter uses the current user directory as the root.
<code>-d root-directory-for-output</code>	Sets the root directory for output.

TABLE 5-4 Converter Command Line Arguments

Option	Description
-debug	Generates the optional debug component of a CAP file. If the -mask option is also specified, the file debug.msk will be generated in the output directory. Note - To generate the debug component, you must first compile your class files with the Java compiler's -g option.
-exportmap	Uses the token mapping from the pre-defined export file of the package being converted. The Converter will look for the export file in the exportpath.
-exportpath <i>list-of-directories</i> >	Specifies the root directories in which the Converter will look for export files. The separator character for multiple paths is the semicolon (;). If this option is not specified, the Converter sets the export path to the Java classpath.
-i	Instructs the Converter to support the 32-bit integer type.
-mask	Cannot be used with -out [CAP]. Indicates this package is for a mask, so restrictions on native methods are relaxed. If you have a source release, you can specify this option to generate a mask out of this package using maskgen.
-nobanner	Suppresses all banner messages.
-noverify	Suppresses the verification of input and output files. For more information on file verification, see “Verification of Input and Output Files” on page 64 .
-nowarn	Instructs the Converter not to report warning messages.
-out [CAP] [EXP] [JCA]	Cannot be used with -mask. Instructs the Converter to output the CAP file, and/or the export file, and/or the Java Card Assembly file. By default (if this option is not specified), the Converter outputs a CAP file and an export file.
-v, -verbose	Enables verbose output. Verbose output includes progress messages, such as “opening file”, “closing file”, and whether the package requires integer data type support.
-V, -version	Prints the Converter version string.
-sign	Specifies to sign the output CAP file
-keystore <i>value</i>	Keystore to use in signing
-storepass <i>value</i>	Keystore password
-alias <i>value</i>	Keystore alias to use in signing
-passkey <i>value</i>	Alias password

TABLE 5-4 Converter Command Line Arguments

Option	Description
<code>-useproxyclass</code>	<p>Cannot be specified with <code>keepproxysource</code>. Builds CAP files as usual in the specified output directory using the existing class files of the application and existing class files of the associated proxy sub-package. New proxy classes are not created.</p> <p>Provides a way for the application developer to build a CAP file with customized proxy files. This option requests the converter to take the class files of the application package and the class files of the co-located proxy sub-package to build a new CAP file. The classes in the application package are converted into new <code>.cap</code> components. New descriptors are created. Dynamically-loaded-classes attributes need to be recomputed based on the new Proxy class file names.</p>
<code>-usecapcomponents</code>	<p>Specifies that the converter retain the specified user supplied CAP components instead of generating them in the final CAP bundle. The input format is as follows:</p> <pre><application classes directory> / <application classes> / javacard/ *.cap</pre>
<code>-keepproxysource <i>directory</i></code>	<p>Cannot be used with <code>-useproxyclass</code>. Creates the proxy source files and other stub files in the specified <i>directory</i>. The converter also builds CAP files as usual in the specified output directory.</p> <p>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).</p>

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_CLASSIC_HOME\api_export_files`, and current directories.

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


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 <configuration file name>
```

The `<configuration file name>` 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 62 were placed in a configuration file, the result would look like this:

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

File Naming for the Converter

This section describes the names of input and output files for the Converter, and gives the correct location for these files. With some exceptions, the Converter follows the Java 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, Classic 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 the 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.

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

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

where `<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:\mywork\java\lang`.

Output File Naming Conventions

The name of the CAP file, export file, and the Java Card Assembly 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 above example, the output files are written by default to the directory `C:\mywork\java\lang\javacard`.

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

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

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

Verification of Input and Output Files

By default, the Converter invokes the Java Card technology-based off-card verifier ("Java Card 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 does 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.

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. (See [“Running the Converter” on page 59](#) and [TABLE 5-4.](#))

Using 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. The Converter also tries to load the shareable interface class files being imported from that package.

For more information on export files, see [Chapter 12](#).

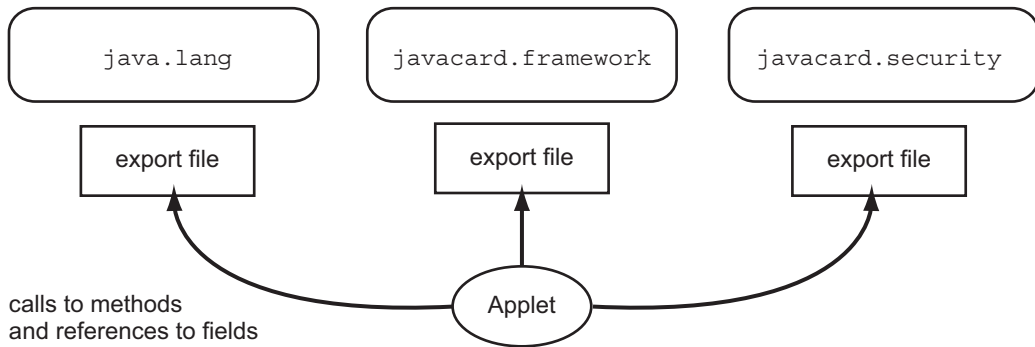
[FIGURE 5-6](#) illustrates how an applet package is linked with the `java.lang`, the `javacard.framework` and `javacard.security` packages via their export files.

You can use the `-exportpath` command option to specify the locations of export files and the shareable interface class files. The path consists of a list of root directories in which the Converter looks for export files and shareable interface class 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 relative directory path that matches the package name. The shareable interface class files are located in the relative directory path that matches the package name.

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

FIGURE 5-6 Calls Between Packages Go Through The Export Files

export files contain mappings to tokens



Specifying an Export Map

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

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 reimplementa-tion).

During package reimplementa-tion, 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, Classic Edition*).

For example, if you have developed a package and would like to reimplement a method (package reimplementa-tion) or upgrade the package by adding new API elements (new exportable classes or new public or protected methods or fields to already existing exportable classes), you must use the `-exportmap` option to preserve binary compatibility with already existing packages that use your package.

The Converter loads the pre-defined export file in the same way that it loads other export files.

Viewing an Export File as Text

The `exp2text` tool is provided to allow you to view any export file in text format (see [TABLE 5-5](#) for a description of the options).

`exp2text.bat [options] package-name`

Note – The file to invoke `exp2text` is a batch file (`exp2text.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

TABLE 5-5 `exp2text` Command Line Options

Option	Description
<code>-classdir input-root-directory</code>	Specifies the root directory where the program looks for the export file.
<code>-d output-root-directory</code>	Specifies the root directory for output.
<code>-help</code>	Prints help message.

If you have a source release, you can localize locale-specific data associated with the `exp2text` tool. For more information, see [Chapter 14](#).

Compatibility for Classic Applets

This chapter describes how to generate application module JAR files from classic applets using the Normalizer tool. These application modules contain classic CAP files and provide compatibility for the Java Card 3 platform by enabling classic applets to run on smart cards with implementations of either the Connected Edition or Classic Edition. See [Chapter 7](#) for more information on CAP files.

Generating Application Modules From Classic Applets

Developers use the Normalizer 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.

The output from the tool is a classic module that contains the class files, the CAP components of the CAP file, SIO proxies for classic SIOs (if required), and associated classic application descriptors. The input to the tool must be classic CAP files and associated export (EXP) files. If the input files are not classic CAP files, the normalization will fail.

Running the Normalizer

The command line interface for the Normalizer has the following syntax (also see [“normalize Subcommands” on page 70](#) in addition of reading this entire section):

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

Note – The file to invoke the Normalizer is a batch file (`normalizer.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

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

- `normalize`
- `copyright` - Displays detailed copyright notice
- `help` - Displays information about the Normalizer command

normalize Subcommands

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 6-6 identifies the `normalize` subcommand options and provides their descriptions.

TABLE 6-6 `normalize` Subcommand Options

Option	Description
<code>-i file</code> or <code>--in file</code>	Specifies the input CAP file name.
<code>-p path</code> or <code>--exportpath path</code>	Specifies the path of the export files used by the tool.
<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.
<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: <directory> / <application classes> / proxy/ [proxy classes] / javacard/ *.cap

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 6-6](#). In this format example, an input file and an output directory are specified as options:

```
normalizer.bat normalize --in file --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
```


Working With CAP Files

One of the files generated by the Converter is the CAP file. The CAP file utilizes the JAR file format, and contains a set of components that describes a Java language package. In addition to the components, the CAP file also contains the manifest file `META-INF/MANIFEST.MF`, which can be used to improve distribution.

This chapter also describes how you can generate a CAP file from a given Java Card Assembly file using the `capgen` tool, and how you can produce an ASCII representation of a CAP file using the `capdump` tool.

Note – The `capgen` and `capdump` tools work only with CAP files generated with versions 2.2.2 and earlier of the Java Card development kit. This chapter contains a sample of the syntax used in CAP files as they apply to version 2.2.2 CAP files and related tools. The CAP file syntax has been updated in the Development Kit version 3.0.2 and this chapter deals primarily with the `capgen` and `capdump` tools and the earlier manifest file format that is still valid and supported with the 3.0.2 Classic Edition, but with these caveats in mind regarding using `capgen` and `capdump`.

CAP File v2.2.2 Manifest File Syntax

A CAP file utilizes the JAR file format, and contains a set of components that describe a Java language package. In addition to the components, the CAP file also contains the manifest file `META-INF/MANIFEST.MF`. The manifest file provides additional human-readable information regarding the contents of the CAP file and the package that it represents. This information can be used to facilitate the distribution and processing of the CAP file.

The information in the manifest file is presented in name:value pairs. These name:value pairs are described in [TABLE 7-7](#).

TABLE 7-7 Name:Value Pairs in the MANIFEST.MF File

Name	Value
Java-Card-CAP-Creation-Time	Creation time of CAP file. For example: Tue Jan 15 11:07:55 PST 2006 The format of the time stamp is operating system-dependent.
Java-Card-Converter-Version	The version of the converter tool. For example: 1.3.
Java-Card-Converter-Provider	Provider of the converter tool. For example: Sun Microsystems, Inc.
Java-Card-CAP-File-Version	CAP file <i>major.minor</i> version. For example: 2.1.
Java-Card-Package-Version	The <i>major.minor</i> version of package. For example: 1.0
Java-Card-Package-AID	AID for the package. For example: 0xa0:0x00:0x00:0x00:0x62: 0x03:0x01:0x0c:0x07
Java-Card-Package-Name	The fully-qualified package name in dot (.) format. For example: javacard.framework
Java-Card-Applet-<n>-AID	The AID for applet <i>n</i> . For example: 0xa0:0x00:0x00:0x00:0x62: 0x03:0x01:0x0c:0x07:0x05
Java-Card-Applet-<n>-Name	Simple class name for applet <i>n</i> . For example: MyApplet
Java-Card-Import-Package-<n>-AID	The AID for imported package <i>n</i> . For example: 0xa0:0x00:0x00:0x00:0x62: 0x00:0x01
Java-Card-Import-Package-<n>-Version	The <i>major.minor</i> version of imported package <i>n</i> . For example: 1.0
Java-Card-Integer-Support-Required	Can be TRUE or FALSE. The value is TRUE if the package requires integer support.

The properties in the manifest file include:

- The names Java-Card-Applet-<n>-AID and Java-Card-Applet-<n>-Name refer to the same applet.
- The converter assigns numbers for the Java-Card-Applet-<n>-NAME and Java-Card-Applet-<n>-AID names in sequential order, beginning with 1.
- The names Java-Card-Imported-Package-<n>-AID and Java-Card-Imported-Package-<n>-Version refer to the same package.
- The converter assigns numbers for the Java-Card-Imported-Package-<n>-AID and Java-Card-Imported-Package-<n>-AID names in sequential order, beginning with 1.

Sample Manifest File

The following code sample illustrates the manifest file that the Converter generates when it converts package jcard.applications. This package contains two applets, MyClass1 and MyClass2.

```
Manifest-Version: 1.0

Created-By: 1.3.1 (Sun Microsystems Inc.)


Java-Card-CAP-Creation-Time: Tue Jan 15 11:07:55 PST 2006

Java-Card-Converter-Version: 1.3

Java-Card-Converter-Provider: Sun Microsystems, Inc.

Java-Card-CAP-File-Version: 2.1

Java-Card-Package-Version: 1.0

Java-Card-Package-Name: jcard.applications

Java-Card-Package-AID: 0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07

Java-Card-Applet-1-Name: MyClass1

Java-Card-Applet-1-AID:
0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x05

Java-Card-Applet-2-Name: MyClass2

Java-Card-Applet-2-AID:
0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x06

Java-Card-Imported-Package-1-AID: 0xa0:0x00:0x00:0x00:0x62:0x00:0x01:0x0c:0x07:0x05

Java-Card-Imported-Package-1-Version: 1.0
```

Java-Card-Imported-Package-2-AID: 0xa0:0x00:0x00:0x00:0x62:0x01:0x01
Java-Card-Imported-Package-2-Version: 1.1
Java-Card-Integer-Support-Required: TRUE

Generating a CAP File From a Java Card Assembly File

Use the `capgen` tool to generate a CAP file from a given Java Card Assembly file. The CAP file that is generated has the same contents as a CAP file produced by the Converter. The `capgen` tool is a backend to the Converter.

Running capgen

Command line syntax for `capgen` is as follows (see [TABLE 7-8](#) for a description of the options):

`capgen.bat [options] filename`

Note – The file to invoke `capgen` is a batch file (`capgen.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

TABLE 7-8 `capgen` Command Line Options

Option	Description
<code>-help</code>	Prints a help message.
<code>-nobanner</code>	Suppresses all banner messages.
<code>filename</code>	Specifies the Java Card Assembly file.
<code>-o filename</code>	Allows you to specify an output file. If the output file is not specified with the <code>-o</code> flag, output defaults to the file <code>a.jar</code> in the current directory.
<code>-version</code>	Outputs the version information.

Producing a Text Representation of a CAP File

Use the `capdump` tool to produce an ASCII representation of a CAP file.

If you have a source release, you can localize locale-specific data associated with the `capdump` tool. For more information, see [Chapter 14](#).

Running `capdump`

Command line usage of `capdump` is as follows where *filename* is the CAP file and there are no command line options, and output from the command is always written to standard output:

```
capdump.bat filename
```

Note – The file to invoke `capdump` is a batch file (`capdump.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

Packaging and Deploying Your Application

This chapter describes how to prepare your applet application to be put into module JAR files, and then to a smart card. The off-card installer, the `scriptgen` tool, resides on your desktop and operates as a packager.

If you have developed your classic application using the Connected Edition of the development kit and are bringing your finished CAP file back into this classic development kit for packaging and deployment, the `scriptgen` tool described in this chapter is where you need to start.

The output from `scriptgen` goes to `apdutool`, which resides on your desktop and acts as a deployment tool. The on-card installer resides in the RE on the card and receives Application Protocol Data Unit commands (APDUs) from `apdutool`.

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 16](#)). They are also used by all RMI samples included with this development kit (See [Chapter 4](#)).

The development kit installer can be used to:

- Dynamically download a Java Card technology package to a Java Card technology-compliant smart card. During development, the CAP file can be installed in the Java Card RE rather than on a Java Card technology-compliant smart card. The installer is capable of downloading version 2.1, 2.2, 2.2.1, 2.2.2, 3.0, 3.0.1 and 3.0.2 Java Card technology-based CAP files.
- Perform necessary on-card linking.
- Delete applets and packages from a Java Card technology-compliant smart card. Once the installer is selected, requests for deletion can be sent from the terminal to the Java Card technology-compliant smart card in the form of APDU commands. For more information, see [“Using the On-card Installer for Deletion” on page 99](#).
- Setting default applets on different logical channels.

The on-card installer is not a multiselectable application. On startup, the on-card installer is the default applet on logical channel 0. The default applet on the other logical channels is set to No applet selected.

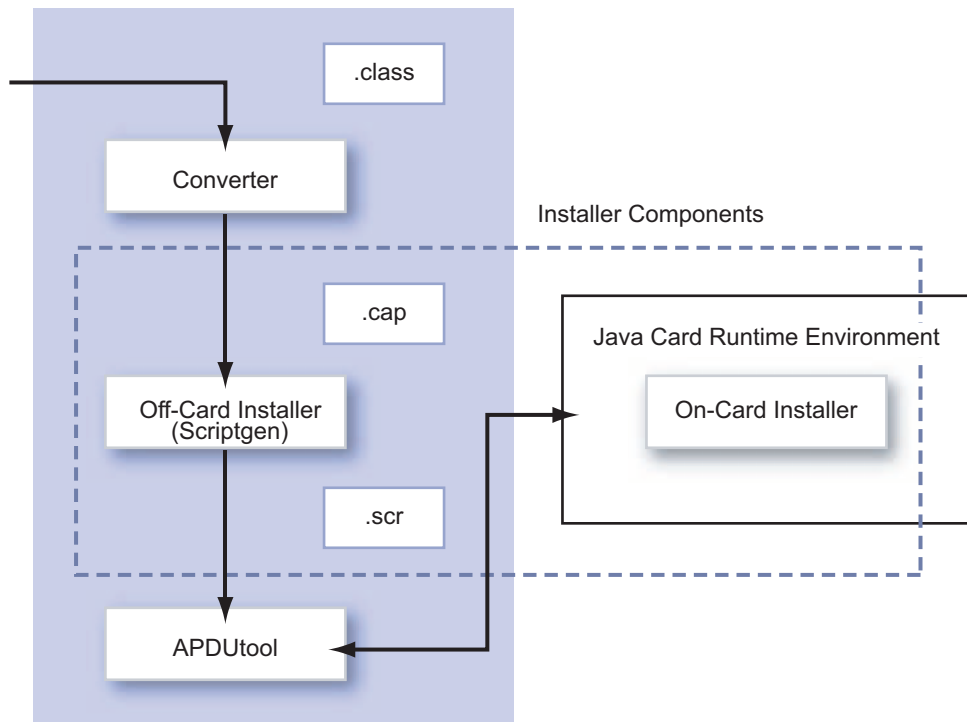
Installer Components and Data Flow

FIGURE 8-7 illustrates the components of the installer and how they interact with other parts of Java Card technology. The dotted line encloses the installer components.

The off-card installer is `scriptgen`. The on-card installer resides on the smart card. `apdutool` is not considered an installer, but processes the output from `scriptgen` and sends it to the on-card installer.

For more information about the installer, see the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*.

FIGURE 8-7 Installer Components



The data flow of the installation process is as follows:

1. The `scriptgen` off-card installer takes as input a version 2.1, 2.2, 2.2.1, 2.2.2, 3.0, 3.0.1 or 3.0.2 CAP file produced by the Converter, and produces a text file that contains a sequence of APDU commands.
2. This set of APDUs is then read by `apdutool` and sent to the on-card installer.
3. The on-card installer processes the CAP file contents contained in the APDU commands as it receives them.
4. The response APDU from the on-card installer contains a status and optional response data.

Running `scriptgen`

The `scriptgen` tool converts a package contained in a CAP file into a script file. The script file contains a sequence of APDUs in ASCII format suitable for another tool, such as `apdutool`, to send to the CAD. The CAP file component order in the APDU script is identical to the order recommended by the *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*.

If you have a source release, you can localize locale-specific data associated with the `scriptgen` tool. For more information, see [Chapter 14](#).

Enter the `scriptgen` command on the command line in this format (see [TABLE 8-9](#) for a description of the options):

```
scriptgen.bat [options] cap-file
```

Note – The file to invoke `scriptgen` is a batch file (`scriptgen.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

TABLE 8-9 `scriptgen` Command Line Options

Option	Description
<code>-help</code>	Prints a help message and exits.
<code>cap-file</code>	File name of the CAP including the full absolute path.
<code>-nobanner</code>	Suppresses printing of the version number.
<code>-nobeginend</code>	Suppresses the output of the “CAP Begin” on page 90” and “CAP End” on page 91” APDU commands.

TABLE 8-9 scriptgen Command Line Options (Continued)

Option	Description
<code>-o filename</code>	Specifies an output filename (default is <code>stdout</code>).
<code>-package package-name</code>	Specifies the name of the package contained in the CAP file. According to the <i>Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition</i> , the CAP file can contain components besides the ones required by the package. This option helps to avoid any possible ambiguity in determining which components should be included.
<code>-version</code>	Prints the version number and exits.

Note – If the CAP file contains components of multiple packages, you must use the `-package <package_name>` option to specify which package to process.

Note – The `apdutil` commands: `powerup`; and `powerdown`; are not included in the output from `scriptgen`.

Sending and Receiving APDUs

The `apdutil` reads a script file containing APDUs and sends them to the RI, or to another Java Card RE. Each APDU is processed and returned to `apdutil`, which displays both the command and response APDUs on the console. Optionally, `apdutil` can write this information to a log file.

If you have a source release, you can localize messages from `apdutil`. For more information, see [Chapter 14](#).

Running apdutil

You run `apdutil` with the following command syntax (see [TABLE 8-10](#) for a description of the options):

```
apdutil.bat [-t0] [-verbose] [-nobanner] [-noatr] [-d | --descriptiveoutput] [-k]
            [-o output-file] [-h host-name] [-p port-number] [-s serial-port ] [-version]
            [-mi] [input-file-name]
```

Note – The file to invoke `apdutool` is a batch file (`apdutool.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

`apdutool` starts listening to APDUs in `T=1` as the default format, unless otherwise specified, on the TCP/IP port specified by the `-p portNumber` parameter for contacted and `portNumber+1` for contactless. The default port is 9025.

TABLE 8-10 `apdutool` Command Line Options

Option	Description
<code>-help</code>	Displays online help for the command.
<code>-h host-name</code>	Specifies the host name on which the TCP/IP socket port is found. (See the flag <code>-p</code> .)
<code>-d</code> or <code>-descriptiveoutput</code>	Formats the output in more user-readable form.
<code>-k</code>	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.
<code>-noatr</code>	Suppresses outputting an ATR (answer to reset).
<code>-nobanner</code>	Suppresses all banner messages.
<code>-o output-file</code>	Specifies an output file. If an output file is not specified with the <code>-o</code> flag, output defaults to standard output.
<code>-p port-number</code>	Specifies a TCP/IP socket port other than the default port (which is 9025).
<code>-s serial-port</code>	<p>Specifies the serial port to use for communication, rather than a TCP/IP socket port. For example, <code>serialPort</code> can be <code>COM1</code>.</p> <p>To use this option, the <code>javax.comm</code> package must be installed on your system. For more information on installing this package, see “Prerequisites to Installing the Development Kit” on page 11.</p> <p>If you enter the name of a serial port that does not exist on your system, <code>apdutool</code> will respond by printing the names of available ports.</p>
<code>-t0</code>	Runs <code>T=0</code> single interface.
<code>-verbose</code>	If enabled, enables verbose <code>apdutool</code> output.

TABLE 8-10 apdutool Command Line Options

Option	Description
-version	Outputs the version information.
-mi	Required if the APDU script is using contacted and contactless commands multiple times in the same script file and the script switches between contacted and contactless interfaces many times.
<i>input-file-name</i>	Specifies an input script file.

apdutool Examples

The following examples show how to use apdutool.

Directing Output to the Console

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

```
apdutool example.scr
```

Directing Output to a File

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

```
apdutool -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-APDUs. Script file commands and C-APDUs are terminated with a semicolon (;). Comments can be of any of the three Java programming language style comment formats (`//`, `/*`, or `/**`).

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

C-APDU syntax for apdutool is as follows:

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

where:

<CLA> :: ISO 7816-4 class byte.
<INS> :: ISO 7816-4 instruction byte.
<P1> :: ISO 7816-4 P1 parameter byte.
<P2> :: ISO 7816-4 P2 parameter byte.
<LC> :: ISO 7816-4 input byte count. 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.

TABLE 8-11 describes each supported script file command in detail noting that they are not case sensitive.

TABLE 8-11 Supported APDU Script File Commands

Command	Description
Note - All APDU script file commands are not case-sensitive.	
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.
select <i>AID</i> ;	Selects the applet with the specified AID, where <i>AID</i> identifies the applet to be selected in the form of //aid/A005453412/151146712. For example: select //aid/A000000062/03010C0101;

TABLE 8-11 Supported APDU Script File Commands

Command	Description
open channel [<i>channel-no</i>] [on <i>origin-channel</i>];	Opens the channel with the channel number specified by <i>channel-no</i> on the origin channel specified by <i>origin-channel</i> , where <i>channel-no</i> is an integer. The default value for the origin channel is basic channel number 0. <i>channel-no</i> and <i>origin-channel</i> are both optional. <i>origin-channel</i> must be an integer from 0-19.
close channel <i>channel-no</i> [on <i>origin-channel</i>];	Closes the channel having the channel number specified by <i>channel-no</i> on origin channel <i>origin-channel</i> , where <i>channel-no</i> is an integer. on <i>origin-channel</i> is optional and the default value for <i>origin-channel</i> is basic channel number 0. <i>origin-channel</i> must be an integer from 0-19.
send APDU [to <i>AID</i>] [on <i>origin-channel</i>];	Sends the APDU specified by <i>APDU</i> after selecting the applet specified by <i>AID</i> on the specified origin channel, where the <i>APDU</i> format uses the C-APDU syntax of the apdutool. on <i>origin-channel</i> is optional and specifies the origin channel to select an applet and send the specified APDU on. The default origin channel is 0 and possible values are 0 - 19. to <i>AID</i> is also optional, and when specified it builds and sends the select command before sending the APDU.

APDUScript Preprocessor Commands

APDUScript 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;
```


Setting Default Applets

The RI supports setting distinct default applets on distinct logical channels and distinct interfaces. This request can be used to set the default applet for a particular logical channel in the specified interface. The applet being set as default must be properly registered with the RI prior to issuing this command.

TABLE 8-12 Set Default Applets on Different Logical Channels

0x8x 0xc6 0xXX 0xYY	Lc: AID length	Data: Default applet AID	Le: ignored
---------------------	----------------	--------------------------	-------------

NOTATION:

- XX is the channel number where the specified applet is configured as default.
- YY is the interface ID where the applet will be configured as default (0 is primary contacted or only interface, 1 is secondary contactless on dual interface).
- AID is the AID of the applet being set as the default.

On-Card Installer Applet AID

The on-card installer applet AID is:
0xa0, 0x00, 0x00, 0x00, 0x62, 0x03, 0x01, 0x08, 0x01.



Downloading CAP Files and Creating Applets

The procedures for CAP file download and applet instance creation are described in the following sections, as are the on-card installer APDU protocol events and APDU types.

Downloading the CAP File

In this procedure, the CAP file is downloaded but applet creation (instantiation) is postponed until a later time. Follow these steps to perform this installation:

1. Use `scriptgen` to convert a CAP file to an APDU script file.

2. Prepend these commands to the APDU script file:

```
powerup;

// Select the installer applet

0x00 0xA4 0x04 0x00 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x08
0x01 0x7F;
```

3. Append this command to the APDU script file:

```
powerdown;
```

4. Invoke apdutool with this APDU script file path as the argument.

Creating an Applet Instance

In this procedure, the applet from a previously downloaded CAP file or an applet compiled in the mask is created. For example, follow these steps to create the JavaPurse applet:

1. Determine the applet AID.

2. Create an APDU script similar to this:

```
powerup;

// Select the installer applet

0x00 0xA4 0x04 0x00 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x08
0x01 0x7F;

// create JavaPurse

0x80 0xB8 0x00 0x00 0x0b 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01
0x04 0x01 0x00

0x7F;

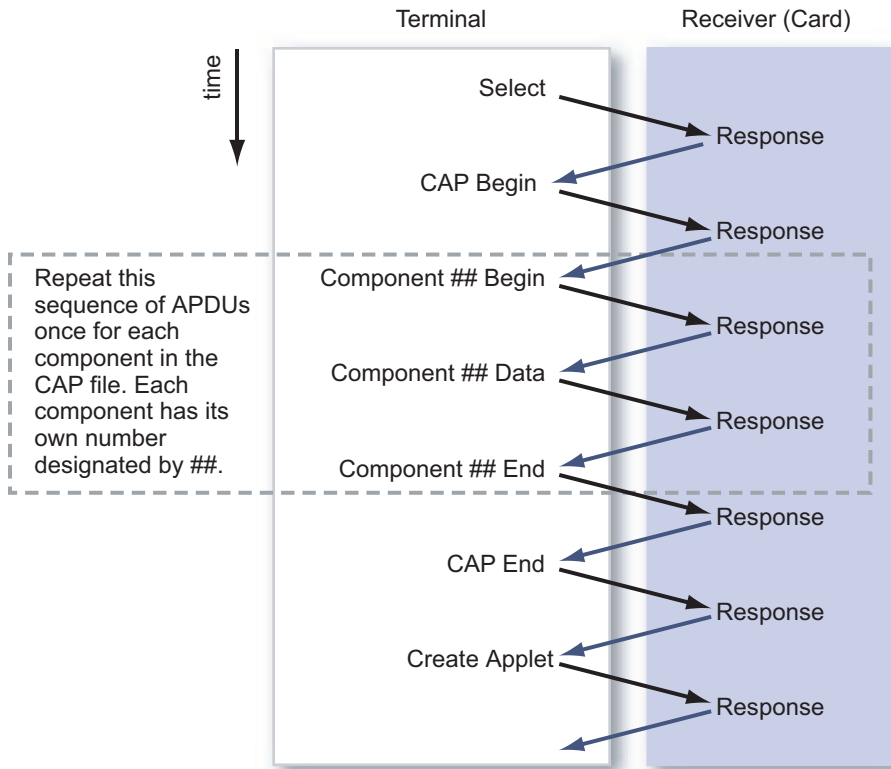
powerdown;
```

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

On-card Installer APDU Protocol

The on-card installer APDU protocol follows a specific time sequence of events in the transmission of Applet Protocol Data Units as shown in [FIGURE 8-8](#).

FIGURE 8-8 On-card Installer APDU Transmission Sequence



APDU Types

There are many different APDU types, which are distinguished by their fields and field values. The following sections describe these APDU types in more detail, including their bit frame formats, field names and field values.

- **Select**
- **Response**
- **CAP Begin**
- **CAP End**
- **Component ## Begin**
- **Component ## End**
- **Component ## Data**
- **Create Applet**

■ **Abort**

Note – In the following APDU commands, the x in the second nibble of the class byte indicates that the installer can be invoked on channels 0, 1, or 2. For example, 0x8x.

Select

TABLE 8-13 specifies the field sequence in the *Select* APDU, which is used to invoke the on-card installer.

TABLE 8-13 *Select* APDU Command

0x0x, 0xa4, 0x04, 0x00	Lc field	Installer AID	Le field
------------------------	----------	---------------	----------

Response

TABLE 8-14 specifies the field sequence in the *Response* APDU. A *Response* APDU is sent as a response by the on-card installer after each APDU that it receives. The *Response* APDU can be either an Acknowledgment (called an ACK), which indicates that the most recent APDU was received successfully, or it can be a Negative Acknowledgement (called a NAK), which indicates that the most recent APDU was not received successfully and must be either resent or the entire installer transmission must be restarted. The first ACK indicates that the on-card installer is ready to receive. The value for an ACK frame SW1SW2 is 9000, and the value for a NAK frame SW1SW2 is 6XXX.

TABLE 8-14 *Response* APDU Command

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

CAP Begin

TABLE 8-15 specifies the field sequence in the *CAP Begin* APDU. The *CAP Begin* APDU is sent to the on-card installer, and indicates that the CAP file components are going to be sent next, in sequentially numbered APDUs.

TABLE 8-15 *CAP Begin* APDU Command

0x8x, 0xb0, 0x00, 0x00	[Lc field]	[optional data]	Le field
------------------------	------------	-----------------	----------

CAP End

TABLE 8-16 specifies the field sequence in the CAP End APDU. The CAP End APDU is sent to the on-card installer, and indicates that all of the CAP file components have been sent.

TABLE 8-16 CAP End APDU Command

0x8x, 0xba, 0x00, 0x00	[Lc field]	[optional data]	Le field
------------------------	------------	-----------------	----------

Component ## Begin

TABLE 8-17 specifies the field sequence in the Component ## Begin APDU. The double pound sign indicates the component token of the component being sent. The CAP file is divided into many components, based on class, method, etc. The Component ## Begin APDU is sent to the on-card installer, and indicates that component ## of the CAP file is going to be sent next.

TABLE 8-17 Component ## Begin APDU Command

0x8x, 0xb2, 0x##, 0x00	[Lc field]	[optional data]	Le field
------------------------	------------	-----------------	----------

Component ## End

TABLE 8-18 specifies the field sequence in the Component ## End APDU. The Component ## End APDU is sent to the on-card installer, and indicates that component ## of the CAP file has been sent.

TABLE 8-18 Component ## End APDU Command

0x8x, 0xbc, 0x##, 0x00	[Lc field]	[optional data]	Le field
------------------------	------------	-----------------	----------

Component ## Data

TABLE 8-19 specifies the field sequence in the Component ## Data APDU. The Component ## Data APDU is sent to the on-card installer, and contains the data for component ## of the CAP file.

TABLE 8-19 Component ## Data APDU Command

0x8x, 0xb4, 0x##, 0x00	Lc field	Data field	Le field
------------------------	----------	------------	----------

Create Applet

TABLE 8-20 specifies the field sequence in the Create Applet APDU. The Create Applet APDU is sent to the on-card installer, and tells the on-card installer to create an applet instance from each of the already sequentially transmitted components of the CAP file.

TABLE 8-20 Create Applet APDU Command

0x8x, 0xb8, 0x00, 0x00	Lc field	AID length field	AID field	parameter length field	[parameters]	Le field
------------------------	----------	------------------	-----------	------------------------	--------------	----------

Abort

TABLE 8-21 specifies the data sequence in the Abort APDU. The Abort APDU indicates that the transmission of the CAP file is terminated, and that the transmission is not complete and must be redone from the beginning in order to be successful.

TABLE 8-21 Abort APDU Command

0x8x, 0xbe, 0x00, 0x00	Lc field	[optional data]	Le field
------------------------	----------	-----------------	----------

APDU Responses to Installation Requests

The installer sends a response code of 0x9000 to indicate that a command completed successfully. Version 3.0.2 of the RI provides a number of codes that can be sent in response to unsuccessful installation requests. [TABLE 8-22](#) describes these codes.

TABLE 8-22 APDU Responses to Installation Requests

Response Code	Description
0x6402	Invalid CAP file magic number. <ul style="list-style-type: none">• Cause: An incorrect magic number was specified in the CAP file.• Solution: Refer to the <i>Java Virtual Machine Specification</i> for the correct magic number. Ensure that the CAP file is built correctly, run it through <code>scriptgen</code>, and download the resulting script file to the card.
0x6403	Invalid CAP file minor number. <ul style="list-style-type: none">• Cause: An invalid CAP file minor number was specified in the CAP file.• Solution: Refer to the <i>Java Virtual Machine Specification</i> for the correct minor number. Ensure that the CAP file is built correctly, run it through <code>scriptgen</code>, and download the resulting script file to the card.
0x6404	Invalid CAP file major number. <ul style="list-style-type: none">• Cause: An invalid CAP file major number was specified in the CAP file.• Solution: Refer to the <i>Java Virtual Machine Specification</i> for the correct major number. Ensure that the CAP file is built correctly, run it through <code>scriptgen</code>, and download the resulting script file to the card.
0x640b	Integer not supported. <ul style="list-style-type: none">• Cause: An attempt was made to download a CAP file that requires integer support into a CREF that does not support integers.• Solution: Either change the CAP file so that it does not require integer support or build the version of CREF that supports integers.
0x640c	Duplicate package AID found. <ul style="list-style-type: none">• Cause: A duplicate package AID was detected in CREF.• Solution: Choose a new AID for the package to be installed.
0x640d	Duplicate Applet AID found. <ul style="list-style-type: none">• Cause: A duplicate Applet AID was detected in CREF.• Solution: Choose a new AID for the applet to be installed.

TABLE 8-22 APDU Responses to Installation Requests

Response Code	Description
0x640f	<p>Installation aborted.</p> <ul style="list-style-type: none"> • Cause: Installation was aborted by an outside command. • Solution: Restart the CAP installation from the beginning and check the <code>INS</code> bytes in the installation script for the offending command.
0x6421	<p>Installer in error state.</p> <ul style="list-style-type: none"> • Cause: A non-recoverable error previously occurred. • Solution: Scan the <code>apduTool</code> output for previous APDU responses indicating an error. Restart the CAP installation.
0x6422	<p>CAP file component out of order.</p> <ul style="list-style-type: none"> • Cause: Installer unable to proceed because it did not receive a component that is a prerequisite to process the current component. • Solution: Check the script file contents for the correct component ordering.
0x6424	<p>Exception occurred.</p> <ul style="list-style-type: none"> • Cause: General purpose error in the installer or applet code. • Solution: Check your applet code for errors.
0x6425	<p>Install APDU command out of order.</p> <ul style="list-style-type: none"> • Cause: Installer APDU commands were received out of order. • Solution: Check the script file for the order of APDU commands. See “On-card Installer APDU Transmission Sequence” on page 89 for more information on the ordering of APDU commands.
0x6428	<p>Invalid component tag number.</p> <ul style="list-style-type: none"> • Cause: An incorrect component tag number was detected during download. • Solution: Refer to Chapter 6 in the <i>Java Virtual Machine Specification</i> for the correct tag number.
0x6436	<p>Invalid install instruction.</p> <ul style="list-style-type: none"> • Cause: An invalid Installer APDU command was received. • Solution: Check the script file for the offending command. See “On-card Installer APDU Transmission Sequence” on page 89 for more information on APDU commands.
0x6437	<p>On-card package max exceeded.</p> <ul style="list-style-type: none"> • Cause: Package installation failed because the number of packages that can be stored on the card has been exceeded. • Solution: Remove some packages from the CREF.
0x6438	<p>Imported package not found.</p> <ul style="list-style-type: none"> • Cause: A package that is required by the current package was not found. • Solution: Download the required package first.

TABLE 8-22 APDU Responses to Installation Requests

Response Code	Description
0x643a	On-card applet package max exceeded. <ul style="list-style-type: none">• Cause: Installation of an applet package failed because the number of applet packages that can be stored on the card has been exceeded.• Solution: Remove some applet packages from the CREF.
0x6442	Maximum allowable package methods exceeded. <ul style="list-style-type: none">• Cause: The limit of 128 package methods on the card has been exceeded.• Solution: Modify the package to support fewer methods.
0x6443	Applet not found for installation. <ul style="list-style-type: none">• Cause: An attempt was made to create an applet instance, but the applet code was not installed on the card.• Solution: Verify that the applet package has been downloaded to the card.
0x6444	Applet creation failed. <ul style="list-style-type: none">• Cause: A general purpose error to indicate that an unsuccessful attempt was made to create the applet.• Solution: Verify availability of resources on the card, check the applet's <code>install</code> method, and so on.
0x644f	Package name is too long. <ul style="list-style-type: none">• Cause: The package name exceeds the length specified in Section 2.2.4.1 of the <i>Java Virtual Machine Specification</i>.• Solution: Replace the name and rebuild.
0x6445	Maximum allowable applet instances exceeded. <ul style="list-style-type: none">• Cause: Creation of the applet instance failed because the number of applet instances that can be stored on the card has been exceeded.• Solution: Remove some applet instances from the CREF.
0x6446	Memory allocation failed. <ul style="list-style-type: none">• Cause: The amount of memory available on the card has been exceeded.• Solution: Verify the amount of memory that is available on the card. Remove packages, applets, and so on, to create enough space. Check the memory requirements of the applet or package being installed or downloaded.
0x6447	Imported class not found. <ul style="list-style-type: none">• Cause: A class that is required by the current class was not found.• Solution: Download the required class first.

A Sample APDU Script

The following is a sample APDU script to download, create, and select the HelloWorld applet.

```
powerup;

// Select the on-card installer applet

0x00 0xA4 0x04 0x00 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01
0x7F;

// CAP Begin

0x80 0xB0 0x00 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Header.cap

// component begin

0x80 0xB2 0x01 0x00 0x00 0x7F;

// component data

0x80 0xB4 0x01 0x00 0x16 0x01 0x00 0x13 0xDE 0xCA 0xFF 0xED 0x01 0x02
0x04 0x00 0x01 0x09 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x0C 0x01 0x7F;

// component end

0x80 0xBC 0x01 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Directory.cap

0x80 0xB2 0x02 0x00 0x00 0x7F;

0x80 0xB4 0x02 0x00 0x20 0x02 0x00 0x1F 0x00 0x13 0x00 0x1F 0x00 0x0E
0x00 0x0B 0x00 0x36 0x00 0x0C 0x00 0x65 0x00 0x0A 0x00 0x13 0x00 0x00
0x00 0x6C 0x00 0x00 0x00 0x00 0x00 0x01 0x7F;

0x80 0xB4 0x02 0x00 0x02 0x01 0x00 0x7F;

0x80 0xBC 0x02 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/Import.cap
```

```

0x80 0xB2 0x04 0x00 0x00 0x7F;

0x80 0xB4 0x04 0x00 0x0E 0x04 0x00 0x0B 0x01 0x00 0x01 0x07 0xA0 0x00
0x00 0x00 0x62 0x01 0x01 0x7F;

0x80 0xBC 0x04 0x00 0x00 0x7F;


// com/sun/javacard/samples/HelloWorld/javacard/Applet.cap

0x80 0xB2 0x03 0x00 0x00 0x7F;

0x80 0xB4 0x03 0x00 0x11 0x03 0x00 0x0E 0x01 0x0A 0xA0 0x00 0x00 0x00
0x62 0x03 0x01 0x0C 0x01 0x01 0x00 0x14 0x7F;

0x80 0xBC 0x03 0x00 0x00 0x7F;


// com/sun/javacard/samples/HelloWorld/javacard/Class.cap

0x80 0xB2 0x06 0x00 0x00 0x7F;

0x80 0xB4 0x06 0x00 0x0F 0x06 0x00 0x0C 0x00 0x80 0x03 0x01 0x00 0x01
0x07 0x01 0x00 0x00 0x00 0x1D 0x7F;

0x80 0xBC 0x06 0x00 0x00 0x7F;


// com/sun/javacard/samples/HelloWorld/javacard/Method.cap

0x80 0xB2 0x07 0x00 0x00 0x7F;

0x80 0xB4 0x07 0x00 0x20 0x07 0x00 0x65 0x00 0x02 0x10 0x18 0x8C 0x00
0x01 0x18 0x11 0x01 0x00 0x90 0x0B 0x87 0x00 0x18 0x8B 0x00 0x02 0x7A
0x01 0x30 0x8F 0x00 0x03 0x8C 0x00 0x04 0x7A 0x7F;

0x80 0xB4 0x07 0x00 0x20 0x05 0x23 0x19 0x8B 0x00 0x05 0x2D 0x19 0x8B
0x00 0x06 0x32 0x03 0x29 0x04 0x70 0x19 0x1A 0x08 0xAD 0x00 0x16 0x04
0x1F 0x8D 0x00 0x0B 0x3B 0x16 0x04 0x1F 0x41 0x7F;

0x80 0xB4 0x07 0x00 0x20 0x29 0x04 0x19 0x08 0x8B 0x00 0x0C 0x32 0x1F
0x64 0xE8 0x19 0x8B 0x00 0x07 0x3B 0x19 0x16 0x04 0x08 0x41 0x8B 0x00
0x08 0x19 0x03 0x08 0x8B 0x00 0x09 0x19 0xAD 0x7F;

0x80 0xB4 0x07 0x00 0x08 0x00 0x03 0x16 0x04 0x8B 0x00 0x0A 0x7A 0x7F;

0x80 0xBC 0x07 0x00 0x00 0x7F;


// com/sun/javacard/samples/HelloWorld/javacard/StaticField.cap

```

```

0x80 0xB2 0x08 0x00 0x00 0x7F;

0x80 0xB4 0x08 0x00 0x0D 0x08 0x00 0x0A 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x7F;

0x80 0xBC 0x08 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/ConstantPool.cap

0x80 0xB2 0x05 0x00 0x00 0x7F;

0x80 0xB4 0x05 0x00 0x20 0x05 0x00 0x36 0x00 0x0D 0x02 0x00 0x00 0x00
0x06 0x80 0x03 0x00 0x03 0x80 0x03 0x01 0x01 0x00 0x00 0x00 0x06 0x00
0x00 0x01 0x03 0x80 0x0A 0x01 0x03 0x80 0x0A 0x7F;

0x80 0xB4 0x05 0x00 0x19 0x06 0x03 0x80 0x0A 0x07 0x03 0x80 0x0A 0x09
0x03 0x80 0x0A 0x04 0x03 0x80 0x0A 0x05 0x06 0x80 0x10 0x02 0x03 0x80
0x0A 0x03 0x7F;

0x80 0xBC 0x05 0x00 0x00 0x7F;

// com/sun/javacard/samples/HelloWorld/javacard/RefLocation.cap

0x80 0xB2 0x09 0x00 0x00 0x7F;

0x80 0xB4 0x09 0x00 0x16 0x09 0x00 0x13 0x00 0x03 0x0E 0x23 0x2C 0x00
0x0C 0x05 0x0C 0x06 0x03 0x07 0x05 0x10 0x0C 0x08 0x09 0x06 0x09 0x7F;

0x80 0xBC 0x09 0x00 0x00 0x7F;

// CAP End

0x80 0xBA 0x00 0x00 0x00 0x7F;

// create HelloWorld

0x80 0xB8 0x00 0x00 0x0b 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x03;
0x01 0x00 0x7F;

// Select HelloWorld

0x00 0xA4 0x04 0x00 9 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x03 0x01
0x7F;

```

```
powerdown;
```

Using the On-card Installer for Deletion

The on-card installer in version 3.0.2 of the Java Card 3 Platform, Classic Edition reference implementation provides the ability to delete package and applet instances from the card's memory. Once the on-card installer is selected, it can receive deletion requests from the terminal in the form of APDU commands. Requests to delete an applet or package cannot be sent from an applet on the card. For more information on package and applet deletion, see the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*.

How to Send a Deletion Request

1. Select the on-card installer applet on the card.
2. Send the APDU for the appropriate deletion request to the installer. The requests that you can send are described in the following sections:
 - [Delete Package](#)
 - [Delete Package and Applets](#)
 - [Delete Applets](#)

For information on the responses that the APDU requests can return, see [“APDU Responses to Deletion Requests” on page 101](#).

APDU Requests to Delete Packages and Applets

You can send requests to delete a package, a package and its applets, and individual applets.

Note – In the following APDU commands, the x in the second nibble of the class byte indicates that the installer can be invoked on channels 0, 1, or 2. For example, 0x8x.

Delete Package

In this request, the Data field contains the size of the package AID and the AID of the package to be deleted. [TABLE 8-23](#) shows the format of the Delete Package request and the expected response.

TABLE 8-23 Delete Package Command

0x8x, 0xc0, 0xXX, 0xXX	Lc field	Data field	Le field
------------------------	----------	------------	----------

The value of 0xXX can be any value for the P1 and P2 parameters. The installer will ignore the 0xXX values. An example of a delete package request on channel 1 would be:

```
//Delete Package Request:
```

```
0x81 0xc0 0x00 0x00 0x08 0x07 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x7F;
```

In this example, 0x07 is the AID length and 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 is the package AID.

Delete Package and Applets

This request is similar to the Delete Package command. In this case the package and applets are removed simultaneously. The data field will contain the size of the package AID and the AID of the package to be deleted. [TABLE 8-24](#) shows the format of the Delete Packages and Applets request and the expected response.

TABLE 8-24 Delete Package and Applets Command

0x8x, 0xc2, 0xXX, 0xXX	Lc field	Data field	Le field
------------------------	----------	------------	----------

The value of 0xXX can be any value for the P1 and P2 parameters. The installer will ignore the 0xXX values. An example of a package and applets deletion request on channel 1 would be:

```
//Delete Package And Applets request
```

```
0x81 0xc2 0x00 0x00 0x08 0x07 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x7F;
```

In this example, 0x07 is the AID length and 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 is the package AID.

Delete Applets

In this request, the “#” symbol in the P1 byte indicates the number of applets to be deleted, which can have a maximum value of eight. The Lc field contains the size of the data field. Data field contains a list of AID size and AID pairs. [TABLE 8-25](#) shows the format of the Delete Applet request and the expected response.

TABLE 8-25 Delete Applet Command

0x8x, 0xc4, 0x0#, 0xXX	Lc field	Data field	Le field
------------------------	----------	------------	----------

The value of 0xXX can be any value for the P2 parameter. The installer will ignore the 0xXX values. An example of a applet deletion request on channel 1 would be:

```
//Delete the applet's request for two applets
```

```
0x81 0xc4 0x02 0x00 0x12 0x08 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x12
0x08 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x13 0x7F;
```

In this example, the “#” symbol is replaced with “2” (0x02) indicating that there are two applets to be deleted. The first applet is 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x12 and the second applet is 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x13.

APDU Responses to Deletion Requests

When the on-card installer receives the request from the terminal, it can return any of the responses shown in [TABLE 8-26](#).

TABLE 8-26 APDU Responses to Deletion Requests

Response Code	Description
0x6a86	Invalid value for P1 or P2 parameter. <ul style="list-style-type: none">• Cause: Value for P1 is less than 1 or greater than 8.• Solution: Ensure that the value for P1 is between 1 and 8.
0x6443	Applet not found for deletion. <ul style="list-style-type: none">• Cause: The applet with the specified AID does not exist.• Solution: Check and correct the AID.
0x644b	Package not found. <ul style="list-style-type: none">• Cause: The package with the specified AID does not exist.• Solution: Check and correct the AID.

TABLE 8-26 APDU Responses to Deletion Requests

Response Code	Description
0x644c	<p>Dependencies on package.</p> <ul style="list-style-type: none">• Cause: Package has other packages dependent on it, or there are some object instances of classes belonging to this package residing in memory.• Solution: Determine which packages are dependent and remove them. If there are object instances of classes belonging to this package residing in memory, try the package and applet deletion combination command to remove the package from card memory.
0x644d	<p>One or more applet instances of this package are present.</p> <ul style="list-style-type: none">• Cause: One or more applet instances of this package are present• Solution: Remove the applets first and then try package deletion, or try the package and applet deletion combination command.
0x644e	<p>Package is ROM package.</p> <ul style="list-style-type: none">• Cause: An attempt was made to delete a package in ROM.• Solution: There is no solution to this problem since packages in ROM cannot be deleted.
0x6448	<p>Dependencies on applet.</p> <ul style="list-style-type: none">• Cause: Other applets are using objects owned by this applet.• Solution: Remove references from other applets to this applet's objects, or try to delete the dependent applets along with this applet.

TABLE 8-26 APDU Responses to Deletion Requests

Response Code	Description
0x6449	Internal memory constraints. <ul style="list-style-type: none">• Cause: There is not enough memory available for the intermediate structures required by applet deletion.• Solution: It may not be possible to recover from this error. One possible thing that can be tried in case of multiple applet deletion is to try to delete applets individually.
0x6452	Cannot delete applet; an applet in the same context is currently active on one of the logical channels. <ul style="list-style-type: none">• Cause: An attempt was made to delete an applet while another applet in the same context is currently active on one of the logical channels.• Solution: In the context of the applet that you are attempting to delete, make sure that no applet is selected on any of the logical channels. Then, re-attempt to delete the applet.
0x6700	Invalid value for <code>Lc</code> parameter. <ul style="list-style-type: none">• Cause: In case of package deletion, the value for <code>Lc</code> is less than 6 or greater than 17. In case of applet deletion, the value for <code>Lc</code> is less than 7 or greater than 136.• Solution: Value of <code>Lc</code> in both of these cases depends on the AIDs being passed in the APDU. Make sure the AIDs are correct and value for <code>Lc</code> is between 6 and 16 in case of package deletion and between 7 and 135 in case of applet deletion.

The response has the format shown in [TABLE 8-27](#).

TABLE 8-27 APDU Response Format

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

On-Card Installer Limits

The limits for the on-card installer are as follows.

- The maximum length of the parameter in the applet creation APDU command is 110.
- The maximum number of packages to be downloaded is 32, including up to 16 applet packages.
- The maximum number of applet instances to be created is 16.
- The maximum length of data in the installer APDU commands is 128.
- No on-card CAP file verification is supported.

- All subsequent APDU commands enclosed in a CAP Begin, CAP End APDU pair will continue to fail after an error occurs.
- The maximum number of applets that can be deleted using one command is eight.

Using the Reference Implementation

This chapter describes how to execute the commands to run the Java Card 3 Platform, Classic Edition Reference Implementation (RI), Version 3.0.2. The RI provides a Java Card runtime environment (RE) executable for the Microsoft Windows XP platform, as well as Java Card RE packages and an installer applet.

The RI is a simulator that can be built with a ROM mask, much like a Java Card technology-based implementation for actual field use. It has the ability to simulate persistent memory (EEPROM), and to save and restore the contents of EEPROM to and from disk files. Applets can be installed in the RI, and it performs I/O via a socket interface, simulating the interaction with a card reader implementing the protocols T=1, T=CL, or T=0 for communications with the card reader (CAD).

The Java Card RI supports the following:

- Use of up to 20 logical channels per communication interface
- Integer data type
- Object deletion
- Card reset in case of object allocation during an aborted transaction

In version 3.0.2 of the development kit, the RI is available as a 32-bit implementation that supports the ability to go beyond the 64KB memory access. The official RI supports the protocols T=1 and T=CL in dual concurrent interface mode. This development kit also support the two protocols T=0 and T=1, both in single interface mode. [TABLE 9-28](#) lists the Java Card RE executables by protocol.

Running the RI

The binary versions of the REs available in this development kit are provided in the `JC_CLASSIC_HOME\bin` directory as the executables `cref_t0.exe`, `cref_t1.exe`, and `cref_tdual.exe`. Each binary corresponds to the supported protocols as shown in [TABLE 9-28](#).

TABLE 9-28 Protocols Supported by RE Executables

RE Executable	Supported Protocol
<code>cref_t0.exe</code>	T=0 protocol (single port), as defined in ISO 7816.
<code>cref_t1.exe</code>	T=1 protocol (single port), as defined in ISO 7816.
<code>cref_tdual.exe</code>	T=1 and T=CL (each on separate port). Uses T=1 for the contact protocol, and the T=CL for the contactless version, as defined in ISO 14473. This is the default executable for <code>cref.bat</code> .

Note – The descriptions, command-line syntax, and options in this chapter for running the formal RI (`cref_tdual.exe`) also apply to the additional supplied Java Card REs, `cref_t0.exe` and `cref_t1.exe`. The difference is solely in the supported protocols.

The development kit also includes the executable `cref.bat` file whose default is to run the RI with T=1/T=CL, but it can also run using the other supported protocols according to arguments shown in [TABLE 9-29 “Case Sensitive Command Line Options for `cref.bat`” on page 107](#).

You run `cref` from the `bin` directory as the working directory to execute the RI as follows ([TABLE 9-29](#) lists the command-line options.):

```
cref.bat [options]
```

Note – The file to invoke `cref` is a batch file (`cref.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

The output of the simulation is logged to standard output, which can be redirected to any desired file. The output stream can range from nothing to very verbose, depending on the command line options selected.

TABLE 9-29 Case Sensitive Command Line Options for `cref.bat`

Option	Description
<code>[-t0 -t1 -tdual]</code>	Specifies the RE version to run according to the desired protocol (see TABLE 9-28). Defaults to <code>-tdual</code> to call <code>cref_tdual.exe</code> if not specified.
<code>-b</code>	Dumps a bytecode histogram at the end of the execution.
<code>-e</code>	Displays the program counter and stack when an exception occurs.
<code>-h, -help</code>	Prints a help screen.
<code>-i input-file-name</code>	Specifies a file to initialize EEPROM. There can be no spaces in the file name argument.
<code>-n</code>	Performs a trace display of the native methods that are invoked.
<code>-nobanner</code>	Suppresses the printing of a program banner.
<code>-nomeminfo</code>	Suppresses the printing of memory statistics when execution starts.
<code>-o output-filename</code>	Saves the EEPROM contents to the named file.
<code>-e2p file-name</code>	Specifies the EEPROM file. If the file exists it is read to initialize the EEPROM image. All <code>e2p</code> images are written to this file. This option is the same as combining <code>-i</code> and <code>-o</code> into one.
<code>-p port-number</code>	Connects to a TCP/IP port using the specified <i>port-number</i> .
<code>-contactedport port-number</code>	Connects to a TCP/IP port using the specified <i>port-number</i> .
<code>-s</code>	Suppresses output. Does not create any output unless followed by other flag options.
<code>-t</code>	Performs a line-by-line trace display of the mask's execution.
<code>-version</code>	Prints only the program's version number. Do not execute.
<code>-z</code>	Prints the resource consumption statistics.

Installer Mask

The development kit installer, the Java Card virtual machine interpreter, and the Java Card platform framework are built into the Installer mask. It can be used as-is to load and run applets. Other than the installer, it does not contain any applets.

The RI requires no other files to start proper interpretation and execution of the mask image's Java Card technology-based bytecode.

Obtaining Resource Consumption Statistics

The Java Card RI provides the `-z` command line option for printing resource consumption statistics. This option enables it to print statistics regarding memory usage once at startup and once at shutdown. Although memory usage statistics will vary among Java Card RE implementations, this option provides the applet developer with a general idea of the amount of memory needed to install and execute an applet.

Note – In addition to the command line option, the Java Card API provides programmatic mechanisms for determining resource usage. For more information on these mechanisms, see the `javacard.framework.JCSystem.getAvailableMemory()` method in the *Application Programming Interface, Java Card Platform, Version 3.0.1, Classic Edition*.

Getting Resource Statistics With the PhotoCard Sample

This section uses the PhotoCard sample program to download and install an applet to illustrate using the large address space available in the 32-bit version of the RI (see [“PhotoCard Sample” on page 36](#)). The sample uses the large address space of the smart card's EEPROM memory to store up to four GIF images. Statistics are provided regarding the following resources: EEPROM, transaction buffer, stack usage, clear-on-reset RAM, and clear-on-deselect RAM. The statistics are printed twice, once at RI start up and once when it shuts down.

CODE EXAMPLE 9-1 shows the output obtained by running the PhotoCard sample program from the `bin` directory with the following command line including the `-z` option:

```
JC_CLASSIC_HOME\bin> cref_t dual -z -o e2p
```

This particular example shows the resources used to download and execute a single application. These statistics can also be used to install a set of applications and execute several transactions.

CODE EXAMPLE 9-1 PhotoCard Sample Showing Resource Statistic Output

```
Java Card 3.0.2 C Reference Implementation Simulator (version 0.41)
32-bit Address Space implementation - with cryptography support
T=1 / T=CL Dual interface APDU protocol (ISO 7816-3)
Copyright 2009 Sun Microsystems, Inc. All rights reserved.

Memory configuration
  Type      Base      Size      Max Addr
  RAM       0x0       0x1000   0xffff
  ROM       0x2000   0xe000   0xffff
  E2P       0x10020  0xffe0   0x1ffff

  ROM Mask size =                0xcec9 =          52937 bytes
  Highest ROM address in mask =   0xeec8 =          61128 bytes
  Space available in ROM =        0x1137 =          4407 bytes
EEPROM will be saved in file "e2p"
Mask has now been initialized for use
  0 bytecodes executed.
    Stack size: 00384 (0x0180) bytes,          00000 (0x0000) maximum used
    EEPROM use: 06994 (0x1b52) bytes consumed, 58510 (0xe48e) available
Transaction buffer: 00000 (0x0000) bytes consumed, 03560 (0x0de8) available
Clear-On-Reset RAM: 00000 (0x0000) bytes consumed, 00576 (0x0240) available
Clear-On-Dsel. RAM: 00000 (0x0000) bytes consumed, 00256 (0x0100) available
CREF was powered down.
  132232 bytecodes executed.
    Stack size: 00384 (0x0180) bytes,          00252 (0x00fc) maximum used
    EEPROM use: 09558 (0x2556) bytes consumed, 55946 (0xda8a) available
Transaction buffer: 00000 (0x0000) bytes consumed, 03560 (0x0de8) available
Clear-On-Reset RAM: 00198 (0x00c6) bytes consumed, 00378 (0x017a) available
Clear-On-Dsel. RAM: 00025 (0x0019) bytes consumed, 00231 (0x00e7) available
Temporary memory usage:
  Java stack: 0 bytes
  Clear on Deselect, channel space 0 : 0 bytes
  Clear on Deselect, channel space 1 : 0 bytes
  Clear on Deselect, channel space 2 : 0 bytes
  Clear on Deselect, channel space 3 : 0 bytes
  Clear on Deselect, channel space 4 : 0 bytes
  Clear on Deselect, channel space 5 : 0 bytes
  Clear on Deselect, channel space 6 : 0 bytes
  Clear on Deselect, channel space 7 : 0 bytes
  Clear on Reset: 0 bytes

C:\JCDK3.0.2\bin>cref_tdual -z -i e2p
Java Card 3.0.2 C Reference Implementation Simulator (version 0.41)
32-bit Address Space implementation - with cryptography support
T=1 / T=CL Dual interface APDU protocol (ISO 7816-3)
```

CODE EXAMPLE 9-1 PhotoCard Sample Showing Resource Statistic Output

Copyright 2009 Sun Microsystems, Inc. All rights reserved.

Memory configuration

Type	Base	Size	Max Addr
RAM	0x0	0x1000	0xffff
ROM	0x2000	0xe000	0xffff
E2P	0x10020	0xffe0	0x1ffff

ROM Mask size = 0xcec9 = 52937 bytes

Highest ROM address in mask = 0xeec8 = 61128 bytes

Space available in ROM = 0x1137 = 4407 bytes

EEPROM (0xffe0 bytes) restored from file "e2p"

Using a pre-initialized Mask

0 bytecodes executed.

Stack size: 00384 (0x0180) bytes, 00000 (0x0000) maximum used

EEPROM use: 09558 (0x2556) bytes consumed, 55946 (0xda8a) available

Transaction buffer: 00000 (0x0000) bytes consumed, 03560 (0x0de8) available

Clear-On-Reset RAM: 00198 (0x00c6) bytes consumed, 00378 (0x017a) available

Clear-On-Dsel. RAM: 00025 (0x0019) bytes consumed, 00231 (0x00e7) available

CREF was powered down.

4624835 bytecodes executed.

Stack size: 00384 (0x0180) bytes, 00250 (0x00fa) maximum used

EEPROM use: 56993 (0xdea1) bytes consumed, 08511 (0x213f) available

Transaction buffer: 00000 (0x0000) bytes consumed, 03560 (0x0de8) available

Clear-On-Reset RAM: 00198 (0x00c6) bytes consumed, 00378 (0x017a) available

Clear-On-Dsel. RAM: 00125 (0x007d) bytes consumed, 00131 (0x0083) available

Temporary memory usage:

Java stack: 0 bytes

Clear on Deselect, channel space 0 : 0 bytes

Clear on Deselect, channel space 1 : 0 bytes

Clear on Deselect, channel space 2 : 0 bytes

Clear on Deselect, channel space 3 : 0 bytes

Clear on Deselect, channel space 4 : 0 bytes

Clear on Deselect, channel space 5 : 0 bytes

Clear on Deselect, channel space 6 : 0 bytes

Clear on Deselect, channel space 7 : 0 bytes

Clear on Reset: 0 bytes

RI Limits

- The maximum number of remote references that can be returned during one card session is 8.

- The maximum number of remote objects that can be exported simultaneously is 16.
 - The maximum number of parameters of type array that can be used in remote methods is 8.
 - The maximum number of Java Card API packages that the Java Card RI can support is 32.
 - The maximum number of library packages that a Java Card system can support is 32.
 - The maximum number of applets that a Java Card system can support is 16.
-

Input and Output

The RI performs I/O via a socket interface, simulating the interaction with a card reader implementing T=1, T=CL, or T=0 communications with the card reader.

Use `apdutool` to read script files and send APDUs via a socket to the RI. See [“apdutool Examples” on page 84](#) for details. Note that you can have the Java Card RI running on one workstation and run `apdutool` on another workstation.

Working With EEPROM Image Files

You can save the state of EEPROM contents, then load it in a later invocation of the RI. To do this, specify an EEPROM image or “store” file to save the EEPROM contents.

Use the `-i` and `-o` flags to manipulate EEPROM image files at the `cref` command line:

- The `-i` flag, followed by a file name, specifies the initial EEPROM image file that will initialize the EEPROM portion of the virtual machine before Java Card virtual machine bytecode execution begins.
- The `-o` flag, followed by a file name, saves the updated EEPROM portion of the virtual machine to the named file, overwriting any existing file of the same name.

The `-i` and `-o` flags do not conflict with the performance of other option flags.

The commit of EEPROM memory changes during the execution of the Java Card RI is not affected by the `-o` flag. Neither standard nor error output is written to the output file named with the `-o` option.

Input EEPROM Image File

The following example shows how the `-i` flag can be used in a useful execution scenario.

```
cref -i e2save
```

The RI attempts to initialize simulated EEPROM from the EEPROM image file named `e2save`. No output file will be created.

Output EEPROM Image File

The following example shows how the `-o` flag can be used in a useful execution scenario.

```
cref -o e2save
```

The Java Card RI writes EEPROM data to the file `e2save`. The file will be created if it does not currently exist. Any existing EEPROM image file named `e2save` is overwritten.

Same Input and Output EEPROM Image File

The following example shows how the `-o` and `-i` flag can be used in a useful execution scenario

```
cref -i e2save -o e2save
```

The Java Card RI attempts to initialize simulated EEPROM from the EEPROM image file named `e2save`, and during processing, saves the contents of EEPROM to `e2save`, overwriting the contents. This behavior is much like a real Java Card technology-compliant smart card in that the contents of EEPROM are persistent.

Different Input and Output EEPROM Image Files

The following example shows how the `-o` and `-i` flag can be used in a useful execution scenario

```
cref -i e2save_in -o e2save_out
```

The RI attempts to initialize simulated EEPROM from the EEPROM image file named `e2save_in`, and during processing, writes EEPROM updates to a EEPROM image file named `e2save_out`. The output file will be created if it does not exist.

Using different names for input and output EEPROM image files eliminates much potential confusion. This command line can be executed multiple times with the same results.

Note – Be careful when naming your EEPROM image files. The Java Card RI will overwrite an existing file specified as an output EEPROM image file. This can, of course, cause a problem if there is already an identically named file with a different purpose in the same directory.

The Default ROM Mask

The Default ROM Mask Version 3.0.2 of the RI provides protocol-related versions of a 32-bit Java Card RE executable (`cref_t dual.exe`) for the Microsoft Windows XP platform, as well as Java Card RE packages and an installer applet.

Producing a Mask File from Java Card Assembly Files

This chapter describes how to use the `maskgen` tool to create a mask from Java Card Assembly files. The `maskgen` tool is not available or of use outside of a source release bundle, so you can disregard this chapter if you do not have a source release of the development kit. If you have a source release, you can localize locale-specific data associated with the `maskgen` tool, see [Chapter 14](#).

The `maskgen` tool produces a mask file from a set of Java Card Assembly files produced by the Converter. The format of the output mask file is targeted to a specific platform. The plug-ins that produce each different `maskgen` output format are called generators. The supported generators are `cref`, which supports the Java Card RE, and `size`, which reports size statistics for the mask. Other generators that are not supported in this release include `jref`, which supports the Java programming language Java Card RE, and `a51`, which supports the Keil A51 assembly language interpreter.

For more information on the contents of a Java Card Assembly file, see [Appendix A](#).

Running `maskgen`

You invoke `maskgen` at the command line as follows (see [TABLE 10-30](#) for a description of the options):

```
maskgen [options] generator filename [filename ...]
```

Note – The file to invoke the `maskgen` is a batch file (`maskgen.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

TABLE 10-30 Command Line Arguments for the maskgen Tool

Argument	Description
<code>-help</code>	Prints a help message.
<code>generator</code>	<p>Specifies the generator, the plug-in that formats the output for a specific target platform. The generators are:</p> <ul style="list-style-type: none"> • <code>a51</code> - Output for the Keil A51 assembly language interpreter (not supported for this release). • <code>cref</code> - Output for the <code>cref</code> interpreter. • <code>jref</code> - Output for the Java programming language Java Card RE interpreter (not supported for this release). • <code>size</code> - Outputs mask size statistics. <p>In this release, the only supported generator is <code>cref</code>.</p>
<code>filename [filename...]</code>	<p>Any number of Java Card Assembly files can be input to maskgen as a whitespace-separated list.</p> <p>You can also create a text file containing a list of Java Card Assembly file names for a new mask, and prepend an “@” character to the name of this text file as an argument to maskgen.</p>
<code>-c filename</code>	<p>Specifies a configuration file, which contains generator-specific settings. For example, the following line maps a native Java Card API method to a native label:</p> <pre>javacard/framework/JCSystem/beginTransaction()V=beginTransaction_NM</pre> <p><code>cref_mask.cfg</code> is an example of a maskgen configuration file.</p>
<code>-debuginfo</code>	Generates debug information for the generated mask. This option is available only with the <code>jref</code> generator.
<code>-nobanner</code>	Suppresses all banner messages.
<code>-o filename</code>	Specifies the file name output from maskgen. If the output file is not specified, output defaults to <code>a.out</code> .
<code>-version</code>	Prints the version number of maskgen, then exits.

Order of Packages on the Command Line

The Java Card Assembly files that can be listed on the command line can belong to API packages, the installer package, or the user's library and applet packages. The Java Card Assembly files that belong to API packages must be listed first on the command line, followed by the Java Card Assembly files belonging to any applets.

If you include the installer package's Java Card Assembly file on the command line, it must be listed after all of the Assembly files belonging to API packages and before the Assembly files of any other applet packages.

For example:

```
maskgen -nobanner cref API_package_1.jca .... API_package_n.jca
      installer_package.jca applet_package_1.jca ...
      applet_package_n.jca
```

Version Numbers for Processed Packages

The packages that you specify to generate a mask can import other packages. These imported packages must share the same major and minor version number as the specified packages.

For example, presume that you are using Package A, version 1.1 to create a mask, and that Package A imports Package B, version 1.1. Then you must ensure that Package B, version 1.1 is listed in the import component of the Package A .jca file.

maskgen Example

This example uses a text file (`args.txt`) to pass command line arguments to `maskgen`:

```
maskgen -o mask.c cref @args.txt
```

where the contents of the file `args.txt` is:

```
first.jca second.jca third.jca
```

This is equivalent to the command line:

```
maskgen -o mask.c cref first.jca second.jca third.jca
```

This command produces an output file `mask.c` that is compiled with a C compiler to produce `mask.o`, which is linked with the Java Card RE interpreter. Refer to [Chapter 9](#) for more information about this target platform.

Building a Custom RI From Sources

This chapter only applies if you have the source release. The `src` folder contains all the files that are only in the source release, including virtual machine (VM) code and all tools. The contents of the `src` folder are described in [“Contents of the Source Release” on page 18](#).

You can modify the RI by adding or modifying its code. The RI consists of `.java` and `C` source files. The core VM is written in the `C` programming language and the rest of the API and supported implementation is written in the Java programming language.

You can modify or add to these files and build a customized Java Card 3 platform RI according to specific requirements. You might want to build a custom RI for the following reasons:

- Add additional classes or packages if a proprietary API or other implementation classes are used.
- Fine tune the existing sources.
- Update the tools to work with a target platform.
- Mask an application into `cref`.

Steps for Building a Custom RI

Before building a custom RI, the software listed in [“Prerequisites to Installing the Development Kit” on page 11](#) must be installed on the system.

The basic steps detailed later in this section to create a custom RI are as follows:

1. Converting your packages using the Converter tool and generating Java Card Assembly files, see [Chapter 5](#).

For example, the output Java Card Assembly (`.jca`) files could be located at:

`.\api_export_files\com\sun\javacard\installer\javacard\installer.jca`

or

`JC_CLASSIC_HOME\samples\classic_applets\HelloWorld\applet\build\classes\com\sun\jcclassic\samples\helloworld\javacard\helloworld.jca`

2. Adding the location for the Java Card Assembly files for your new packages in these files:

- `src\vm_16.in`
- `src\vm_32.in`

This step is required for new packages. If you are modifying existing packages, this step is not required.

3. Building the `cref` executable.

See [“Building the 32-Bit Custom RI” on page 120](#), and if necessary [“Building the 16-Bit Custom RI” on page 121](#).

The new `cref` is built with the new packages masked in. Masked applications can be instantiated without requiring download after the runtime environment starts up.

4. Testing the custom RI.

See [“Testing the 32-Bit Custom RI” on page 121](#) and if necessary [“Building the 16-Bit Custom RI” on page 121](#).

▼ Building the 32-Bit Custom RI

Note – The following procedure builds the 32-bit version of the RI. To build the 16-bit version first see [“Building the 16-Bit Custom RI” on page 121](#).

1. Edit the files or add more files.
2. Update the tools source code, if required.
3. From the command line, navigate to the `src` folder and run the `ant all` command.

The `ant all` command creates the `JC_CLASSIC_HOME\custom_build` folder with a `bin` and `lib` folder under it.

The `custom_build\bin` directory contains the new `cref` and all of the other `.bat` files for the tools.

The `custom_build\lib` folder contains the `.jar` files and configuration files.

▼ Testing the 32-Bit Custom RI

1. Run the new `cref` file stored in `JC_CLASSIC_HOME\custom_build\bin` noting the expected console output in [CODE EXAMPLE 11-1](#) and its specific reference in the content to the 32-bit Address Space.

```
JC_CLASSIC_HOME\custom_build\bin\cref_tdual.exe [options]
```

See [Chapter 9](#) for a description of the available options for `cref`.

Files created as a result of running or building the custom RI are stored in the `JC_CLASSIC_HOME\custom_build\bin` and `JC_CLASSIC_HOME\custom_build\lib` directories. These directories are created the first time the custom RI is built and are over-written every time the custom RI is built.

2. Run `apdutool` in a separate window to send in your `apdu` scripts to `cref`.

```
apdutool -nobanner -noatr filename.scr > filename.scr.cref.out
```

For information on `apdutool`, see [“Running apdutool” on page 82](#). If the run is successful, the `apdutool.log`, `filename.scr.cref.out`, is identical to the file `filename.scr.expected.out`.

CODE EXAMPLE 11-1 Expected Console Output When Building 32-Bit Custom RI

```
Java Card 3.0.2 C Reference Implementation Simulator
32-bit Address Space implementation - with cryptography support
T=1 / T=CL Dual interface APDU protocol (ISO 7816-4)
Copyright (c) 2009 Sun Microsystems, Inc. All rights reserved.
Use is subject to license terms.

Memory configuration
  Type      Base      Size      Max Addr
  RAM       0x0       0xc78    0xc77
  ROM       0x2000    0xc800   0xe7ff
  E2P       0x10020   0xffe0   0x1ffff

  ROM Mask size =                0x8b3e =          35646 bytes
  Highest ROM address in mask =   0xab3d =          43837 bytes
  Space available in ROM =        0x3cc2 =          15554 bytes

Mask has now been initialized for use
```

▼ Building the 16-Bit Custom RI

By default the procedure described in [“Building the 32-Bit Custom RI” on page 120](#) builds the 32-bit `cref`. To build `cref` with 16-bit support you must modify those steps as follows:

Note – Building the 16-bit version creates only one `cref_t0.exe` file with no `t1` or `tdual` version, so the resulting `cref.bat` file executes only `cref_t0.exe`.

1. Before starting, clean any previous builds by running `ant clean`.
2. When executing `ant all`, set this property: `for.bit=16`, with the following modified command:

`ant -Dfor.bit=16 all`
3. When testing the 16-bit build, execute `cref.bat` from the `custom_build/bin` directory and watch for the expected output depicted in [CODE EXAMPLE 11-2](#) noting its specific reference in the content to the 16-bit Address Space.

CODE EXAMPLE 11-2 Expected Console Output When Building 16-Bit Custom RI

```
Java Card 3.0.2 C Reference Implementation Simulator
16-bit Address Space implementation - with cryptography support
T=0 Extended APDU protocol (ISO 7816-3)
Copyright (c) 2009 Sun Microsystems, Inc. All rights reserved.
Use is subject to license terms.

Memory configuration
  Type      Base      Size      Max Addr
  RAM       0x0       0x600    0x5ff
  ROM       0x600    0xbc00   0xc1ff
  E2P       0xc200   0x3be0   0xfddf

  ROM Mask size =                0x9fb0 =      40880 bytes
  Highest ROM address in mask =   0xa5af =      42415 bytes
  Space available in ROM =        0x1c50 =       7248 bytes
Mask has now been initialized for use
```

Verifying CAP and Export Files

Off-card verification provides a means for evaluating CAP and export files in a desktop environment. When applied to the set of CAP files that will reside on a Java Card technology compliant smart card and the set of export files used to construct those CAP files, the Java Card technology-enabled off-card verifier provides the means to assert that the content of the smart card has been verified.

If you have a source release, you can localize locale-specific data associated with the off-card verifier. For more information, see [Chapter 14](#).

The off-card verifier is a combination of three tools, `verifycap`, `verifyexp`, and `verifyrev`. The following sections describe how to use each tool.

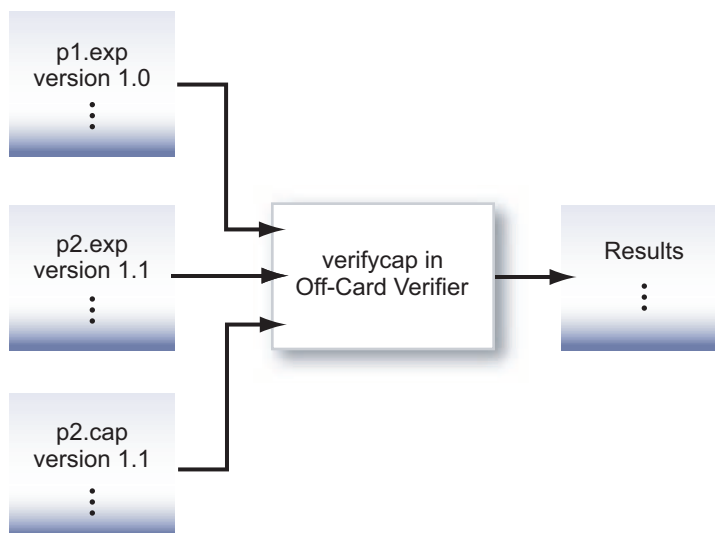
- `verifycap` - see [“Verifying CAP Files” on page 123](#).
- `verifyexp` - see [“Verifying Export Files” on page 125](#).
- `verifyrev` - see [“Verifying Binary Compatibility” on page 126](#).

Verifying CAP Files

The `verifycap` tool is used to verify a CAP file within the context of package's export file (if any) and the export files of imported packages. This verification confirms whether a CAP file is internally consistent, as defined in Chapter 6 of the *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*, and consistent with a context in which it can reside in a Java Card technology-enabled device.

Each individual export file is verified as a single unit. The scenario is shown in [FIGURE 12-9](#). In the figure, the package `p2` CAP file is being verified. Package `p2` has a dependency on package `p1`, so the export file from package `p1` is also input. The `p2.exp` file is only required if `p2.cap` exports any of its elements.

FIGURE 12-9 Verifying a CAP file



Running `verifycap`

You invoke `verifycap` at the command line as follows (see [TABLE 12-31](#) for a description of options):

```
verifycap.bat [options] export-files CAP-file
```

Note – The file to invoke `verifycap` is a batch file (`verifycap.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

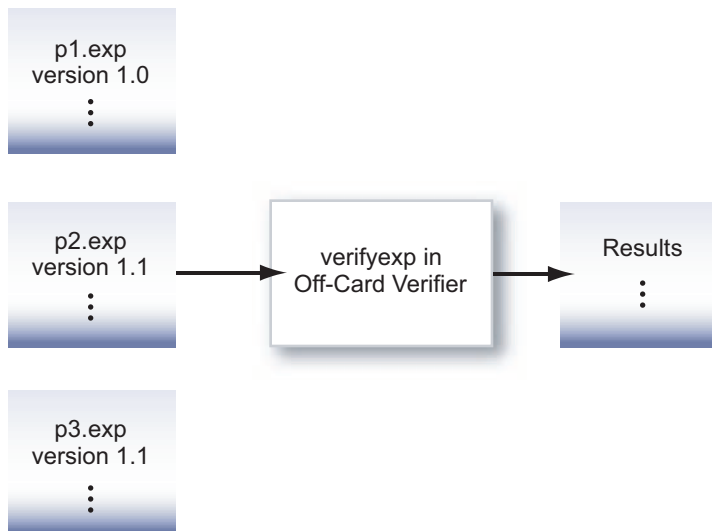
TABLE 12-31 `verifycap` Command Line Arguments

Argument	Description
<i>export-files</i>	A list of export files of the packages that this CAP file uses.
<i>CAP-file</i>	Name of the CAP file to be verified.
	For more <code>verifycap</code> options, also see “Command Line Options for Off-Card Verifier Tools” on page 127.

Verifying Export Files

The `verifyexp` tool is used to verify an export file as a single unit. This verification is “shallow,” examining only the content of a single export file, not including export files of packages referenced by the package of the export file. The verification determines whether an export file is internally consistent and viable as defined in Chapter 5 of the *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*. This scenario is illustrated in [FIGURE 12-10](#).

FIGURE 12-10 Verifying An Export File



Running `verifyexp`

You invoke `verifyexp` at the command line as follows (see [TABLE 12-32](#) for a description of options):

```
verifyexp [options] export-file
```

Note – The file to invoke `verifyexp` is a batch file (`verifyexp.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

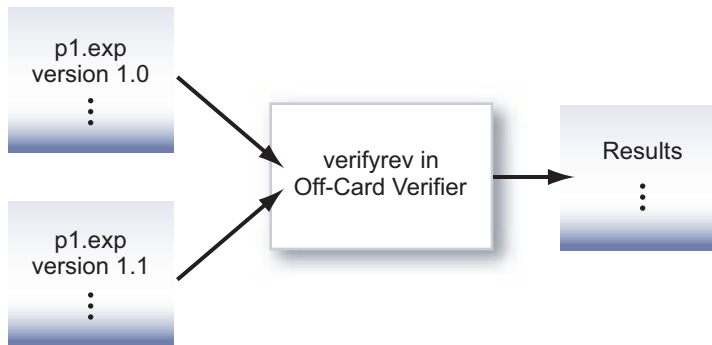
TABLE 12-32 verifyexp Command Line Argument

Argument	Description
<export file>	Fully qualified path and name of the export file. For more verifyexp options, also see “ Command Line Options for Off-Card Verifier Tools ” on page 127.

Verifying Binary Compatibility

The `verifyrev` tool checks for binary compatibility between revisions of a package by comparing the respective export files. This scenario is illustrated in [FIGURE 12-11](#). The export files from version 1.0 and 1.1 of package `p1` are input to `verifyrev`. The verification examines whether the Java Card platform version rules, including those imposed for binary compatibility as defined in Section 4.4 of the *Virtual Machine Specification, Java Card Platform, Version 3.0.1, Classic Edition*, have been followed.

FIGURE 12-11 Verifying Binary Compatibility Of Export Files



Running `verifyrev`

You invoke `verifyrev` at the command line as follows (see [“Command Line Options for Off-Card Verifier Tools”](#) on page 127 for more options in addition to those described in this section):

```
verifyrev.bat [options] export-file export-file
```

Note – The file to invoke `verifyrev` is a batch file (`verifyrev.bat`) that must be run from a working directory of `JC_CLASSIC_HOME\bin` in order for the code to execute properly.

The first *export-file* argument on the command line represents the fully qualified path of the export files to be compared, while the second export file name must be the same as the first one with a different path, for example:

```
verifyrev d:\testing\old\crypto.exp d:\testing\new\crypto.exp
```

Command Line Options for Off-Card Verifier Tools

The `verifycap`, `verifyexp`, and `verifyrev`, off-card verifier tools share many of the same command line options. The only exception is the `-package` option which is available for `verifycap` only.

These options exhibit the same behavior regardless of the tool that calls them.

TABLE 12-33 `verifycap`, `verifyexp`, `verifyrev` Command Line Options

Option	Description
<code>-help</code>	Prints help message.
<code>-nobanner</code>	Suppresses banner message.
<code>-nowarn</code>	Suppresses warning messages.
<code>-package <package name></code>	(Available for <code>verifycap</code> only) Sets the name of the package to be verified.
<code>-verbose</code>	Enables verbose mode.
<code>-version</code>	Prints version number and exit.
<code>-C command-options-file</code> or <code>--commandoptionsfile</code> <code>command-options-file</code>	Optional. Specifies a file containing command-line options.

PART II Programming With the Development Kit

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

Using Cryptography Extensions

This chapter describes how to use the basic security and cryptography classes, which do not appear in all Java Card development kits. For the location of cryptography files in the development kit, see [“Installed Files and Directories” on page 17](#).

The security and cryptography classes are supported by the RI (`cref`). The support for security and cryptography allows you to:

- Generate message digests using the SHA1 algorithm
- Generate cryptographic keys on Java Card technology-compliant smart cards for use in the ECC and RSA algorithms
- Set cryptographic keys on Java Card technology-compliant smart cards for use in the AES, DES, 3DES, ECC, and RSA algorithms
- Encrypt and decrypt data with the keys using the AES, DES, 3DES, and RSA algorithms
- Generate signatures using the AES, DES, 3DES, ECC, or SHA and RSA algorithms
- Generate sequences of random bytes
- Generate checksums
- Use part of a message as padding in a signature block

Note – DES is also known as single-key DES. 3DES is also known as triple-DES.

For more information on the SHA1, DES, 3DES, and RSA encryption schemes, see:

- For SHA1—*“Secure Hash Standard”*, FIPS Publication 180-1:
<http://www.itl.nist.gov/>
- For DES—*“Data Encryption Standard (DES)”*, FIPS Publication 46-2 and *“DES Modes of Operation”*, FIPS Publication 81:
<http://www.itl.nist.gov/>
- For RSA—*“RSAES-OAEP (Optional Asymmetric Encryption Padding) Encryption Scheme”*:
<http://www.rsasecurity.com/>

- For AES—*“Advanced Encryption Standard (AES)”* FIPs Publication 197:
<http://www.itl.nist.gov/>
- For ECC—*“Public Key Cryptography for the Financial Industry: The Elliptic Curve Digital Signature Algorithm”* (ECDSA): X9.62-1998
<http://www.x9.org/>
- For Checksum—*“Information technology—Telecommunications and information exchange between systems—High-level data link control (HDLC) procedures”*
ISO/IEC-13239:2002 (replaces ISO-3309):
<http://www.iso.org/>

Supported Cryptography Classes

The implementation of security and cryptography in version 3.0.2 of the RI supports the use of the following classes:

- `javacardx.crypto.Cipher`
- `javacard.security.Checksum`
- `javacard.security.InitializedMessageDigest`
- `javacard.security.KeyAgreement`
- `javacard.security.KeyBuilder`
- `javacard.security.KeyPair`
- `javacard.security.MessageDigest`
- `javacard.security.RandomData`
- `javacard.security.Signature`
- `javacard.security.SignatureMessageRecovery`

Note – In version 3.0.2 of the RI, the implementation of `RandomData` is not suitable for porting.

TABLE 13-34 lists the cryptography algorithms that are implemented for the RI.

TABLE 13-34 Algorithms Implemented by the Cryptography Classes

Class	Algorithm
Checksum	<ul style="list-style-type: none"> • ALG_ISO3309_CRC16—ISO/IEC 3309-compliant 16-bit CRC algorithm. This algorithm uses the generator polynomial: $x^{16}+x^{12}+x^5+1$. The default initial checksum value used by this algorithm is 0. This algorithm is also compliant with the frame-checking sequence as specified in section 4.2.5.2 of the ISO/IEC 13239 specification. • ALG_ISO3309_CRC32—ISO/IEC 3309-compliant 32-bit CRC algorithm. This algorithm uses the generator polynomial: $X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^5+X^4+X^2+X+1$. The default initial checksum value used by this algorithm is 0. This algorithm is also compliant with the frame-checking sequence as specified in section 4.2.5.3 of the ISO/IEC 13239 specification.
Cipher	<ul style="list-style-type: none"> • ALG_DES_CBC_ISO9797_M2—provides a cipher using DES in CBC mode. This algorithm uses CBC for DES and 3DES. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV'96) scheme. • ALG_RSA_PKCS1—provides a cipher using RSA. Input data is padded according to the PKCS#1 (v1.5) scheme. • ALG_AES_BLOCK_128_CBC_NOPAD—provides a cipher using AES with block size 128 in CBC mode and does not pad input data.
InitializedMessageDigest	Provides the functionality of MessageDigest, with the additional ability to allow for initialization with a starting hash value corresponding to a previously hashed part of the message. Provide for SHA1 and SHA256.
KeyAgreement	<ul style="list-style-type: none"> • ALG_EC_SVDP_DH—elliptic curve secret value derivation primitive, Diffie-Hellman version, as per [IEEE P1363]. • ALG_EC_SVDP_DHC—elliptic curve secret value derivation primitive, Diffie-Hellman version, with cofactor multiplication, as per [IEEE P1363].
KeyBuilder	<p>The algorithms define the key lengths for:</p> <ul style="list-style-type: none"> • 128-bit AES • 64-bit DES • 112-, 128-, 160-, 192-bit ECC • 128-bit DES3 • 512-bit RSA
KeyPair	<p>The algorithms define the key lengths for:</p> <ul style="list-style-type: none"> • 112-, 128-, 160-, 192-bit ECC • 512-bit RSA
MessageDigest	Message digest algorithm SHA1 and SHA256

TABLE 13-34 Algorithms Implemented by the Cryptography Classes

Class	Algorithm
RandomData	Pseudo-random number generator with a 48-bit seed, which is modified using a linear congruential formula.
Signature	<ul style="list-style-type: none"> • <code>ALG_DES_MAC8_ISO9797_M2</code>—generates an 8-byte MAC (most significant 8 bytes of encrypted block) using DES or 3DES in CBC mode. This algorithm uses CBC for DES and 3DES. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV'96) scheme. • <code>ALG_RSA_SHA_PKCS1</code>—encrypts the 20 byte SHA1 digest using RSA. The digest is padded according to the PKCS#1 (v1.5) scheme. • <code>ALG_AES_MAC_128_NOPAD</code>—generates a 16-byte MAC using AES with blocksize 128 in CBC mode and does not pad input data. • <code>ALG_ECDSA_SHA</code>—signs/verifies the 20-byte SHA digest using ECDSA.
SignatureMessageRecovery	<ul style="list-style-type: none"> • <code>ALG_RSA_SHA_ISO9796_MR</code>—This algorithm uses the first part of the input message as padding bytes during signing. During verification, these message bytes (recoverable message) can be recovered to reconstruct the message.

Instantiating the Classes

Implementations of the cryptography classes extend the corresponding base class with implementations of their abstract methods. All data allocation associated with the implementation instance is performed when the instance is constructed. This is done to ensure that any lack of required resources can be flagged when the applet is installed.

Each cryptography class, except `KeyPair`, has a `getInstance` method which takes the desired algorithm as one of its parameters. The method returns an instance of the class in the context of the calling applet. Instead of using a `getInstance` method, `KeyPair` takes the desired algorithm as a parameter in its constructor.

If you request an algorithm that is not listed in [TABLE 13-34](#) or that is not implemented in this release, `getInstance` will throw a `CryptoException` with reason code `NO_SUCH_ALGORITHM`.

Localizing With The Development Kit

This chapter describes the support for localization that the development kit provides. This chapter is useful only if you have a source release of the development kit.

Items that can be localized in the development kit include the Java language-based tools, `cref`, and the Java language-based Java Card RMI sample applications and client framework. The Java language-based programs and the C language-based programs use different localization mechanisms.

Localization Support for Java Utilities

This section describes the mechanisms used to localize the following programs and tools:

- RMI sample programs
- RMI client framework
- `scriptgen`
- `apdutool`
- `converter`
- `normalizer`
- `maskgen`
- `capdump`
- `exp2text`
- off-card verifier

These Java utilities and programs can be localized in a similar fashion. Each uses the Java language resource bundle mechanism. This mechanism allows the user to customize locale-sensitive data for a new locale without rebuilding the application. Refer to the Java SE platform `java.util.ResourceBundle` class for more information regarding resource bundles.

The development kit also provides localization support for Java Card RMI sample applications and client framework. Localizing the client framework and the sample applications can be done in the same way as the Java Card technology-based utilities.

Since none of the Java Card platform reference implementation utilities or programs require a graphical user interface (GUI) and are not dependent on user input, the majority of the locale-specific data consists of static strings. Localization consists of customizing these strings for the intended locale. Locale-sensitive strings are grouped into `.properties` files (for example, `MessagesBundle.properties`). Localizing an application entails creating a new version of the properties file that contains the translated strings.

Localizing a Java Program to a New Locale

The following steps are required to localize a Java program to a new locale:

- 1. Create a new version of the appropriate property file which contains the correct set of strings customized for the intended locale.**
- 2. Rename the property file with the appropriate locale identifier appended to the file name (for example, the French version of the `MessagesBundle.properties` file would be `MessagesBundle_fr.properties`).**
- 3. Include the location of the property file in the CLASSPATH for the `cref` utility.**

When the Java utility is executed in an environment with the same locale as the properties file, the strings contained in that properties file will be used for output.

For additional information regarding internationalization and localization in the Java language, please refer to the Java SE online documentation at <http://java.sun.com/j2se/1.5.0/docs/guide/intl/index.html>.

Localization Support for `cref`

Similar to the Java utilities described above, localizing `cref` consists of providing the set of static output strings used by `cref` correctly customized for the intended locale. Unlike the Java language utilities described above, `cref` must be rebuilt to localize it to a new locale. Therefore, you must use a source bundle of this development kit to localize `cref`.

All of the locale-sensitive strings used by `cref` are stored in a single C header file, `src/share/c/common/cref_locale.h`. To localize `cref`, customize the strings in this file for the new locale and then rebuild `cref`.

Programming to the Java Card RMI Client-Side API

This chapter describes how to use the Java Card RMI client-side API. A Java Card RMI client application runs on a Card Acceptance Device (CAD) terminal that supports a Java SE or Java ME platform. The client application requires a portable and platform-independent mechanism to access the Java Card RMI server applet executing on the smart card. For an example, see [“RMIPurse Sample” on page 38](#)).

For best results use the Java Card RMI client-side API for Java Card RMI client programs. The RI for the classic platform supports the optional Java Cad RMI functionality. The RI for the connected platform does not support the optional Java Cad RMI functionality.

The basic client-side framework is implemented in the package `com.sun.javacard.javacard.rmiclientlib` and `com.sun.javacard.javacard.clientlib`.

The library is located in the file `JC_CLASSIC_HOME\lib\tools.jar`.

The reference implementation of the Java Card RMI client-side API is based on APDU I/O for its card access mechanisms. For more information on APDU I/O, see [Chapter 16](#).

Javadoc tool files for the RMI client-side APIs are located in this bundle at `JC_CLASSIC_HOME\docs\rmiclient`.

Remote Stub Object

The Java Card RMI API supports two formats for passing remote references. The format for remote references containing the class name requires stubs for remote objects available to the client application.

The standard Java RMIC compiler tool can be used as the stub compilation tool to produce stub classes required for the client application. To produce these stub classes, the RMIC compiler tool must have access to all the non-abstract classes defined in the applet package which directly or indirectly implement remote interfaces. In addition, it needs to access the `.class` files of all the remote interfaces implemented by them.

If you want the stub class to be Java Card RMI-specific when it is instantiated on the client, it must be customized with a Java Card platform-specific implementation of the `CardObjectFactory` interface.

The standard Java RMIC compiler is used to generate the remote stub objects. `JCRemoteRefImpl`, a Java Card platform-specific implementation of the `java.rmi.server.RemoteRef` interface, allows these stub objects to work with the Java Card RMI API. The stub object delegates all method invocations to its configured `RemoteRef` instance.

The `com.sun.javacard.rmiclientlib.JCRemoteRefImpl` class is an example of a `RemoteRef` object customized for the Java Card platform.

For examples of how to use these interfaces and classes, see *Application Programming Notes, Java Card Platform, Version 3.0.1, Classic Edition*.

Note – Since the remote object is configured as a Java Card platform-specific object with a local connection to the smart card via the `CardAccessor` object, the object is inherently not portable. A bridge class must be used if it is to be accessed from outside of this client application.

Note – Some versions of the RMIC do not treat `Throwable` as a superclass of `RemoteException`. The workaround is to declare remote methods to throw `Exception` instead.

Java Card RMI Client-Side API

The two packages in the Java Card RMI client-side reference implementation demonstrate remote stub customization using the RMIC compiler generated stubs and card access for Java Card applets.

The package `com.sun.javacard.javacard.rmiclientlib` implements Java Card RMI-specific functionality.

The package `com.sun.javacard.javacard.clientlib` implements basic functionality to exchange APDUs with a smart card or a smart card simulator. This implementation of `clientlib` requires that the `ApuIO` library is included in the CLASSPATH.

In the release bundles, the Javadoc tool files for this API are located under:

```
JC_CLASSIC_HOME\docs\rmiclient
```

Package `rmiclientlib`

This package includes several classes.

- **class `JCRMICConnect`**—The main class of the RMI framework that provides methods to select a card applet and to get an initial reference.
- **class `JCCardObjectFactory`**—An implementation of the `CardObjectFactory` that processes the data returned from the card in the format defined in the *Runtime Environment Specification, Java Card Platform, Version 3.0.1, Classic Edition*. Any object references must contain class names.
- **class `JCCardProxyFactory`**—The `JCCardProxyFactory` class is similar to `JCCardObjectFactory`, but processes references containing lists of names. `JCCardProxyFactory` uses the JDK 1.4.+ proxy mechanism to generate proxies dynamically.
- **class `JCRemoteRefImpl`**—An implementation of interface `java.rmi.server.RemoteRef`. These remote references can work with stubs generated by the RMIC compiler with the `-v1.2` option.

The main method is:

```
public Object invoke(Remote remote, Method method, Object[]  
params, long unused) throws IOException, RemoteException,  
Exception
```

This method prepares the outgoing APDU, passes it to `CardAccessor`, and then uses `CardObjectFactory` to parse the returned APDU and instantiate the returned object or throw an exception.

Package `clientlib`

This package includes an interface and a class.

- **interface `CardAccessor`**—An interface defining methods to exchange APDUs with a card and to close connection to a card.

- **class AduIOCardAccessor**—A simple implementation of the CardAccessor interface that passes the APDUs to a card or a card simulator using the AduIO library. This class takes parameters to start the AduIO from the file `jcclient.properties`, which must be included in CLASSPATH.

Working with APDU I/O

This chapter describes the APDU I/O API, which is a library used by many Java Card development kit components, such as `apdutool`, and the RMI client framework, see [Chapter 15](#).

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 `JC_CLASSIC_HOME\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_CLASSIC_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 exchangeApu(Apu apu)`

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 145](#) 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 (`crcf`) 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 (`crcf`) 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 `crcf`, 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 145](#).

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)
```

Programming for the Large Address Space

This chapter describes two ways in which you can take advantage of large memory storage in smart cards: by using library packages properly and by separating your data properly. This chapter also includes a sample.

The default address space automatically built in the RI is the large address space. Allowing your applications to take advantage of the large address capabilities of the Classic Edition RI requires careful planning and programming. Some size limitations still exist within the reference implementation. The way that you structure large applications, as well as applications that manage large amounts of data, determines how the large address space can be exploited.

Programming Large Applications and Libraries

The key to writing large applications for the Java Card 3 Platform, Classic Edition is to divide the code into individual package units. The most important limitation on a package is the 64KB limitation on the maximum component size. This is especially true for the Method component: if the size of an application's Method component exceeds 64KB, then the Java Card converter will not process the package and will return an error.

You can overcome the component size limitation by dividing the application into separate application and library components. The Java Card platform has the ability to support library packages. Library packages contain code which can be linked and reused by several applications. By dividing the functionality of a given application

into application and library packages, you can increase the size of the components. Keep in mind that there are important differences between library packages and applet packages:

- In a library package, all public fields are available to other packages for linking.
- In an applet package, only interactions through a shareable interface are allowed by the firewall.

Therefore, you should not place sensitive or exclusive-use code in a library package. It should be placed in an applet package, instead.

Handling a Package as a Separate Code Space

Several applications and API functionality can be installed in the smart card simultaneously by handling each package as a separate code space. This technique will let you exceed the 64KB limit, and provide full Java Card API functionality and support for complex applications requiring larger amounts of code.

Storing Large Amounts of Data

The most efficient way to take advantage of the large memory space is to use it to store data. Today's applications are required to securely store ever-growing amounts of information about the cardholder or network identity. This information includes certificates, images, security keys, and biometric and biographical information.

This information sometimes requires large amounts of storage. Before version 2.2.2, versions of the Java Card platform reference implementation had to save downloaded applications or user data in valuable persistent memory space. Sometimes, the amount of memory space required was insufficient for some applications. However, the memory access schemes introduced with version 2.2.2 allow applications to store large amounts of information, while still conforming to the Java Card specification.

The Java Card specification does not impose any requirements on object location or total object heap space used on the card. It specifies only that each object must be accessible by using a 16-bit reference. It also imposes some limitations on the amount of information an individual object is capable of storing, by using the number of fields or the count of array elements. Because of this loose association, it is possible for any given implementation to control how an object's information is stored, and how much data these objects can collectively hold.

The Java Card 3 Platform, Classic Edition reference implementation, allows you to use all of the available persistent memory space to store object information. By allowing you to separate data storage into distinct array and object types, this reference implementation allows you to store the large amounts of data demanded by today's applications.

Example: The photocard Demo Applet

The photocard demo applet, included at `samples/classic_applets/PhotoCard`, is an example of an application that takes advantage of the large address space capabilities.

The photocard applet performs a very simple task: it stores pictures inside the smart card and retrieves them by using a Java Card RMI interface, see [Chapter 15](#). For more information on the photocard demo applet and how to run it, see ["PhotoCard Sample" on page 36](#). The source code is located in the source code bundle at:

```
JC_CLASSIC_HOME\samples\classic_applets\PhotoCard\applet\src\com\
sun\jcclassic\samples\photocard
```

The collection of arrays (more than two arrays would be required in this case) can easily hold far more than 64KB of data. Storing this amount of information should not be a problem, provided that enough mutable persistent memory is configured in the RE.

Java Card Assembly Syntax Example

This appendix contains an annotated Java Card platform assembly (“Java Card Assembly”) file output from the Converter. The comments in this file are intended to aid the developer in understanding the syntax of the Java Card Assembly language, and as a guide for debugging Converter output.

Note – For source releases, use the Java `javadoc` tool with the file `JC_CLASSIC_HOME\src\tools\converter\com\sun\javacard\jcas\Parser.jj` to get an HTML file with the BNF grammar for the Java Card Assembly syntax.

```
/*
 * Java Card Assembly annotated example. The code
 * contained within this example is not an executable
 * program. The intention of this program is to illustrate the
 * syntax and use of the Java Card Assembly directives and commands.
 *
 * A Java Card Assembly file is textual representation of the
 * contents of a CAP file.
 *
 * The contents of a Java Card Assembly file are hierarchically
 * structured. The format of this structure is:
 *
 *     package
 *
 *     package directives
```

```

*      imports block
*
*      applet declarations
*
*      constant pool
*
*      class
*
*          field declarations
*
*          virtual method tables
*
*          interface table
*
*          [remote interface table] - only for remote classes
*
*      methods
*
*          method directives
*
*          method statements
*
*
* Java Card Assembly files support both the Java single line
* comments and Java block
* comments. Anything contained within a comment is ignored.
*
*
* Numbers may be specified using the standard Java notation.
*
* Numbers prefixed
*
* with a 0x are interpreted as
*
* base-16, numbers prefixed with a 0 are base-8, otherwise
*
* numbers are interpreted
*
* as base-10.
*
*/

/*
* A package is declared with the .package directive. Only one
* package is allowed

```

```

* inside a Java Card Assembly
* file. All directives (.package, .class, et.al) are case
* insensitive. Package,
* class, field and
* method names are case sensitive. For example, the .package
* directive may be written
* as .PACKAGE,
* however the package names example and ExAmPle are different.
*/

.package example {

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

.aid 0:1:2:3:4:5:6:7:8:9:0xa:0xb:0xc:0xd:0xe:0xf;

    // the AIDs length must be
    // between 5 and 16 bytes inclusive

.version 0.1;          // major version <DOT> minor version

    /*
    * The imports block declares all of packages that this
    * package imports. The data
    * that is declared
    * in this section appears in the Import Component of the

```

```

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

.imports {
    0xa0:0x00:0x00:0x00:0x62:0x00:0x01 1.0;
    // java/lang aid <SPACE>
    // java/lang major version <DOT> java/lang minor version
    0:1:2:3:4:5 0.1;                // package test2
    1:1:2:3:4:5 0.1;                // package test3
    2:1:2:3:4:5 0.1;                // package test4
}

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

```

```

*/

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

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

```

```

.constantPool {

    /*
     * The first six entries declare constant pool entries that
     * are contained in
     * other packages.
     * Note that superMethodRef are always declared internal
     * entry.
     */

    classRef      0.0;      // 0    package token 0, class token 0
    instanceFieldRef 1.0.2; // 1    package token 1, class token 0,
                                //    instance field token 2
    virtualMethodRef 2.0.2; // 2    package token 2, class token 0,
                                //    instance field token 2
    classRef      0.3;    // 3    package token 0, class token 3
    staticFieldRef 1.0.4;  // 4    package token 1, class token 0,
                                //    field token 4
    staticMethodRef 2.0.5; // 5    package token 2, class token 0,
                                //    method token 5

    /*
     * The next five entries declare constant pool entries
     * relative to this class.
     *
     */

    classRef      test0;      // 6
    instanceFieldRef  test1/field1;      // 7
    virtualMethodRef  test1/method1()V;      // 8
    superMethodRef  test9/equals(Ljava/lang/Object;)Z;      // 9

```



```

        staticFieldRef    test1/field0;           // 10
        staticMethodRef   test1/method3()V;      // 11
    }

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

```

```

.class final public test1 0 extends 0.0 {

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

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

    /*
     * The public method table declares the virtual methods within
     * this classes
     * public virtual method
     * table. The number following the directive is the method
     * table base (See the
     * Class Component specification).
     *
     * Method names declared in this table are relative to
     * this class. This
     * directive is required even if there
     * are not virtual methods in this class. This is necessary
     * to establish the
     * method table base.

```

```

*/

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

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

.packagemethodtable 0 {}

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

.class final public test9 9 extends test1 {

    .publicmethodtable 0 {
        equals(Ljava/lang/Object;)Z;

```

```

        method1()V;

        method2()V;
    }

    .packagemethodtable 0 {}

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

        return;
    }
}

.class final public test0 1 extends 0.0 {

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

    .PublicMethodTable 1 {
        equals(Ljava/lang/Object;)Z;
        abc()V;                // method must be in this class
        def()V;
        labelTest()V;
        instructions()V;
    }
}

```

```

.PackageMethodTable 0 {
    ghi()V;                // method must be in this class
    jkl()V;
}

// if the class implements more than one interface, multiple
// interfaceInfoTables will be present.
.implementedInterfaceInfoTable

.interface 1.0 {    // java/rmi/Remote
}

.interface RemoteAccount { // The table contains method tokens
10; // getBalance()S
9;  // debit(S)V
8;  // credit(S)V
11; // setAccountNumber([B)V
12; // getAccountNumber()[B
}
}

.implementedRemoteInterfaceInfoTable { // The table contains
// method tokens

// excluding java.rmi.Remote
.interface RemoteAccount { // Contains method tokens
getBalance()S    10;    // getBalance()S
debit(S)V        9;    // debit(S)V
credit(S)V       8;    // credit(S)V
setAccountNumber([B)V 11; // setAccountNumber([B)V
getAccountNumber()[B 12; // getAccountNumber()[B

```

```

}

}

/*
 * Declaration of 2 public visible virtual methods and two
 * package visible
 * virtual methods..
 */
.method public abc()V 1 {
    return;
}

.method public def()V 2 {
    return;
}

.method ghi()V 0x80 {
    // per the CAP file
    //specification, method tokens
    // for package visible methods
    return; // must have the most significant bit set to 1.
}

.method jkl()V 0x81 {
    return;
}

/*
 * This method illustrates local labels and exception table
 * entries. Labels
 * are local to each

```

```

* method. No restrictions are placed on label names except
* that they must
* begin with an alphabetic
* character. Label names are case insensitive.
*
* Two method directives are supported, .stack and .locals.
* These
* directives are used to
* create the method header for each method. If a method
* directive is omitted,
* the value 0 will be used.
*
*/

.method public static install([BSB)V 0 {
    .stack 0;
    .locals 0;

10:
11:
12:
13:
14:
15:

    return;

/*
* Each method may optionally declare an
* exception table. The start offset,

```

```

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

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

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

```



```

*/

.method public labelTest()V 3 {

L1:          goto L2;

L2:          goto L1;

            goto_w L1;

            goto_w L3;

L3:          return;
}

/*
 * This method illustrates the use of each Java Card platform
 * instruction for version 3.0.2.
 * Mnemonics are case insensitive.
 *
 * See the Java Card virtual machine specification for
 * the specification of
 * each instruction.

```

```

        */

        .method public instructions()V 4 {

            aaload;

            aastore;

            aconst_null;

        aload 0;
        aload_0;
        aload_1;
        aload_2;
        aload_3;
        anewarray 0;
        areturn;
        arraylength;
        astore 0;
        astore_0;
        astore_1;
        astore_2;
        astore_3;
        athrow;
        baload;
        bastore;
        bipush 0;
        bpush 0;
        checkcast 10 0;
        checkcast 11 0;
        checkcast 12 0;

```

```
checkcast 13 0;
checkcast 14 0;
dup2;
dup;
dup_x 0x11;
getfield_a 1;
getfield_a_this 1;
getfield_a_w 1;
getfield_b 1;
getfield_b_this 1;
getfield_b_w 1;
getfield_i 1;
getfield_i_this 1;
getfield_i_w 1;
getfield_s 1;
getfield_s_this 1;
getfield_s_w 1;
getstatic_a 4;
getstatic_b 4;
getstatic_i 4;
getstatic_s 4;
goto 0;
goto_w 0;
i2b;
i2s;
iadd;
iaload;
iand;
```

```
iastore;

icmp;

iconst_0;

iconst_1;

iconst_2;

iconst_3;

iconst_4;

iconst_5;

iconst_m1;

idiv;

if_acmpeq 0;

if_acmpeq_w 0;

if_acmpne 0;

if_acmpne_w 0;

if_scmpeq 0;

if_scmpeq_w 0;

if_scmpge 0;

if_scmpge_w 0;

if_scmpgt 0;

if_scmpgt_w 0;

if_scmpne 0;

if_scmpne_w 0;

if_scmplt 0;

if_scmplt_w 0;

if_scmpne 0;

if_scmpne_w 0;

ifeq 0;

ifeq_w 0;
```

```
ifge 0;
ifge_w 0;
ifgt 0;
ifgt_w 0;
ifle 0;
ifle_w 0;
iflt 0;
iflt_w 0;
ifne 0;
ifne_w 0;
ifnonnull 0;
ifnonnull_w 0;
ifnull 0;
ifnull_w 0;
iinc 0 0;
iinc_w 0 0;
iipush 0;
iload 0;
iload_0;
iload_1;
iload_2;
iload_3;
ilookupswitch 0 1 0 0;
impdep1;
impdep2;
imul;
ineg;
instanceof 10 0;
```

```
instanceof 11 0;
instanceof 12 0;
instanceof 13 0;
instanceof 14 0;
invokeinterface 0 0 0;
invokespecial 3;    // superMethodRef
invokespecial 5;    // staticMethodRef
invokestatic 5;
invokevirtual 2;
ior;
irem;
ireturn;
ishl;
ishr;
istore 0;
istore_0;
istore_1;
istore_2;
istore_3;
isub;
itableswitch 0 0 1 0 0;
iushr;
ixor;
jsr 0;
new 0;
newarray 10;
newarray 11;
newarray 12;
```

```

newarray 13;

newarray boolean[];           // array types may be declared
numerically or

newarray byte[];             // symbolically.

newarray short[];

newarray int[];

nop;

pop2;

pop;

putfield_a 1;
putfield_a_this 1;
putfield_a_w 1;
putfield_b 1;
putfield_b_this 1;
putfield_b_w 1;
putfield_i 1;
putfield_i_this 1;
putfield_i_w 1;
putfield_s 1;
putfield_s_this 1;
putfield_s_w 1;
putstatic_a 4;
putstatic_b 4;
putstatic_i 4;
putstatic_s 4;

ret 0;

return;

s2b;

s2i;

```

```
sadd;

saload;

sand;

sastore;

sconst_0;

sconst_1;

sconst_2;

sconst_3;

sconst_4;

sconst_5;

sconst_m1;

sdiv;

sinc 0 0;

sinc_w 0 0;

sipush 0;

sload 0;

sload_0;

sload_1;

sload_2;

sload_3;

slookupswitch 0 1 0 0;

smul;

sneg;

sor;

srem;

sreturn;

sshl;

sshr;
```



```

    sspush 0;
    sstore 0;
    sstore_0;
    sstore_1;
    sstore_2;
    sstore_3;
    ssub;
    stableswitch 0 0 1 0 0;
    sushr;
    swap_x 0x11;
    sxor;

    }
}

.class public test2 2 extends 0.0 {

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

```

```

.class public test3 3 extends test2 {

    /*
    * Declaration of static array initialization is done the same way
    * as in Java
    * Only one dimensional arrays are allowed in the
    * Java Card platform
    * Array of zero elements, 1 element, n elements
    */

    .fields {

        public static final int[] array0 0 = {}; // [I
        public static final byte[] array1 1 = {17}; // [B
        public static short[] arrayn 2 = {1,2,3,...,n}; // [S
    }

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

.interface public test4 4 extends 0.0 {
}

```

}

Additional Optional Ant Tasks

The command line tools in this development kit execute Apache Ant transparently, so you are not required to use Ant directly to use the command line tools themselves. Those Ant tasks are required to install and run the development kit.

In addition, this development kit also includes additional, optional Apache Ant tasks for skilled Ant users to streamline using the development kit. These optional Ant tasks grouping several command line tools into a single Ant task. This chapter describes how to use these additional, optional, and unsupported Apache Ant tasks.

Location and Installation

The optional Ant tasks are included at:

`JC_CLASSIC_HOME\lib\jctasks.jar`

These tasks work with Apache Ant version 1.6.5 or later. You can use the `ant -version` command to verify the installed version.

Note – Use of the additional Ant tasks described in this section is strictly optional and is not formally supported by Sun Microsystems, Inc., nor has it been fully tested.

▼ Installing the Ant Tasks

1. Be sure Ant is configured as described in [“Downloading the Development Kit” on page 12](#).

2. Copy the file `JC_CLASSIC_HOME\lib\jctasks.jar` to a directory that serves as your Ant tasks home directory.
3. Add the `jctasks.jar` file to your classpath or put it into the *Ant-Home-Path\lib* directory to be automatically be picked up when Ant is run.

Where:

- *Ant-Home-Path* is the path to the Ant installation.
- The value of the `ANT_HOME` environment variable is properly configured to run Ant (see [“Downloading the Development Kit” on page 12](#)).

▼ Setting Up the Optional Ant Tasks

The following XML must be added your `build.xml` file to use the optional Ant tasks in your build.

```
<!-- Definitions for tasks for Java Card tools -->
<taskdef name="apdutool"
  classname="com.sun.javacard.ant.tasks.APDUToolTask" />
<taskdef name="capgen"
  classname="com.sun.javacard.ant.tasks.CapgenTask" />
<taskdef name="maskgen"
  classname="com.sun.javacard.ant.tasks.MaskgenTask" />
<taskdef name="deploycap"
  classname="com.sun.javacard.ant.tasks.DeployCapTask" />
<taskdef name="exp2text"
  classname="com.sun.javacard.ant.tasks.Exp2TextTask" />
<taskdef name="convert"
  classname="com.sun.javacard.ant.tasks.ConverterTask" />
<taskdef name="verifyexport"
  classname="com.sun.javacard.ant.tasks.VerifyExpTask" />
<taskdef name="verifycap"
  classname="com.sun.javacard.ant.tasks.VerifyCapTask" />
<taskdef name="verifyrevision"
```

```

        classname="com.sun.javacard.ant.tasks.VerifyRevTask" />
<taskdef name="scriptgen"
        classname="com.sun.javacard.ant.tasks.ScriptgenTask" />
<typedef name="appletnameaid"
        classname="com.sun.javacard.ant.types.AppletNameAID" />
<typedef name="jcainputfile"
        classname="com.sun.javacard.ant.types.JCAInputFile" />
<typedef name="exportfiles"
        classname="org.apache.tools.ant.types.FileSet" />

```

Library Dependencies

The libraries from the Java Card development kit in [TABLE B-1](#) are needed in your classpath if you are using the indicated feature. Alternatively, you can specify the classpath nested element for each task to put the required JAR files in the classpath during build execution.

TABLE B-1 Library Dependencies

LIBRARIES	FEATURES
converter.jar and offcardverifier.jar	Creating CAP, EXP or JCA files. Using maskgen to create a mask. Dumping contents of an EXP file in a text file.
offcardverifier.jar	Verifying EXP files, CAP files and verifying binary compatibility between two versions of an export file.
apdutool.jar and apduio.jar	Sending an APDU script to cref.
capdump.jar	Dumping contents of a CAP file.
scriptgen.jar	Generating a APDU script from a CAP file.
apdutool.jar, apduio.jar and scriptgen.jar	Installing a CAP file in cref and generate resulting EEPROM image.

Ant Task Descriptions

The eleven Ant tasks provided in the Ant tasks bundle are provided to simplify the use of the development kit for Ant users. This section describes each of these Ant tasks and how to use them. Note that the JAR files for the tasks are expected to be in the system classpath, unless otherwise noted.

- [“APDUTool” on page 183.](#)
- [“CapDump” on page 185.](#)
- [“Capgen” on page 186.](#)
- [“Converter” on page 188.](#)
- [“DeployCap” on page 191.](#)
- [“Exp2Text” on page 193.](#)
- [“Maskgen” on page 195.](#)
- [“Scriptgen” on page 198.](#)
- [“VerifyCap” on page 199.](#)
- [“VerifyExp” on page 201.](#)
- [“VerifyRev” on page 203.](#)

APDUTool

Runs APDUTool to send the APDU script file to `cref` and check if all APDUs were sent correctly. You can set `CheckDownloadFailure=true` to stop the build if any response status is not 9000.

APDUTool is invoked in a different instance of the Java Virtual Machine¹ software (JVM™ software) than the one being used by Ant.

TABLE B-2 Parameters for APDUTool

Attribute	Description	Required
ScriptFile	Fully qualified path and name of the APDU script file.	Yes
CrefExe	Fully qualified path and name of <code>cref</code> executable.	Yes
OutEEFile	Output EEPROM file that will contain the EEPROM image after <code>cref</code> finishes execution.	Yes
CheckDownloadFailure	Stops the build if any response status coming back from <code>cref</code> is not 9000.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory in which to invoke the JVM software.	No
InEEFile	Input EEPROM file for <code>cref</code> . If specified <code>cref</code> initiates using the EEPROM image stored in this file.	No
nobanner	Set this element to <code>true</code> if you do not want the APDUTool banner showing.	No
version	Prints the version number of APDUTool.	No

Errors

Execution of this task fails if any of the required elements are not supplied, if `apdutool.jar` and `apduio.jar` are not in the classpath, or if APDUTool returns an error code.

1. The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java(TM) platform.

Examples

Runs APDUTool to send APDUs in APDU script file `test.scr` to `cref` and to check if all APDUs were sent correctly. Also checks that the response returned from the card was 9000.

```
<target name="APDUToolTarget" >
    <apdutool
        scriptFile="${samples.helloworld.script}"
        outEEFile="${samples.eeprom}/outEEFile"
        CrefExe="${jcardkit_home}/bin/cref.exe">
    </apdutool>
</target>
```

Run the APDUTool to install the APDU script in `test.scr` file to `cref` and check if the APDU commands were processed successfully. Classpath in this example is referenced by the classpath `refid`.

```
<target name="APDUToolTarget" >
    <apdutool
        scriptFile="${samples.helloworld.script}"
        outEEFile="${samples.eeprom}/outEEFile"
        CheckDownloadFailure="true"
        CrefExe="${jcardkit_home}/bin/cref.exe">
        <classpath refid="classpath"/>
    </apdutool>
</target>
```

Run APDUTool to install the APDU script in `test.scr` file to `cref`, which is initialized using a stored EEPROM image from the file `inEEFile`. Also check if the APDU commands were sent correctly. Classpath used in this example is referenced by the classpath `refid`.

```
<target name="APDUToolTarget" >
    <apdutool
        scriptFile="${samples.helloworld.script}"
        outEEFile="${samples.eeprom}/outEEFile"
        inEEFile="${samples.eeprom}/inEEFile"
        CheckDownloadFailure="true"
        CrefExe="${jcardkit_home}/bin/cref.exe">
        <classpath refid="classpath"/>
    </apdutool>
</target>
```

CapDump

Runs the CapDump tool to dump the contents of a CAP file.

TABLE B-3 Parameters for CapDump

Attribute	Description	Required
CapFile	Fully qualified name of CAP file.	Yes
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory in which to invoke the JVM software.	No

Errors

Execution of this task fails if CapFile element is not supplied, if `capdump.jar` is not in the classpath, or if CapDump returns an error code.

Examples

Run CapDump to dump the contents of the `test.cap` file.

```
<target name="CapDumpTarget" >
  <capdump
    CapFile="${samples.output}/test.cap"
  </capdump>
</target>
```

Run CapDump to dump the contents of the `test.cap` file. Classpath used in this example is referenced by the `classpath` refid

```
<target name="CapDumpTarget" >
  <capdump
    CapFile="${samples.output}/test.cap"
    <classpath refid="classpath"/>
  </capdump>
</target>
```

Capgen

Runs Capgen to generate a CAP file from a JCA file.

TABLE B-4 Parameters for Capgen

Attribute	Description	Required
JCAFile	Fully qualified path and name of the input JCA file.	Yes
OutFile	Fully qualified path and name of the output CAP file.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory in which to invoke the JVM software.	No
nobanner	Set this element to <code>true</code> if you do not want the Capgen banner showing.	No
version	Prints Capgen version number.	No

Errors

Execution of this task fails if any of the required elements are not supplied, if `converter.jar` is not in the classpath, or if Capgen returns an error code.

Examples

Run Capgen to generate the `mathDemo.cap` file from the `mathDemo.jca` file.

```
<target name="CapgenTarget" >
  <capgen
    JCAFile="${sample.output}/mathDemo.jca"
    outfile="${sample.output}/mathDemo.cap">
    <classpath refid="classpath"/>
  </capgen>
</target>
```

Run Capgen to generate a `mathDemo.cap` file from the `mathDemo.jca` file. Classpath used in this example is referenced by the `classpath` refid.

```
<target name="CapgenTarget" >
  <capgen
    JCAFile="${sample.output}/mathDemo.jca"
    outfile="${sample.output}/mathDemo.cap">
```

```
        <classpath refid="classpath"/>
      </capgen>
</target>
```

The following example is the same as the previous example, except no output file is specified. Capgen generates `out.cap` in the directory in which the JVM software was invoked.

```
<target name="CapgenTarget" >
  <capgen
    JCAFile="${sample.output}/mathDemo.jca"/>
    <classpath refid="classpath"/>
  </capgen>
</target>
```

Converter

Runs Converter to generate CAP, EXP and JCA files from a Java technology-based package. By default the Java Card platform converter creates CAP and EXP files for the input package. However, if any one of the CAP, JCA or EXP flags are enabled, only the output files enabled are generated.

TABLE B-5 Parameters for Converter

Attribute	Description	Required
PackageName	Fully qualified name of the package being converted.	Yes
PackageAID	AID of the package being converted.	Yes
MajorMinorVersion	Major and Minor version numbers of the package, for example, 1.2 (where 1 is major version number and 2 is minor version number).	Yes
CAP	If enabled tells the converter to create a CAP file.	No
EXP	If enabled tells the converter to create a EXP file.	No
JCA	If enabled tells the converter to create a JCA file.	No
ClassDir	The root directory of the class hierarchy. Specifies the directory where the converter will look for class files.	No
Int	If enabled turns on support the 32-bit integer type.	No
Debug	If enabled, enables generation of debugging information.	No
ExportPath	Root directories where the Converter will look for export files.	No
ExportMap	If enabled, tells the converter to use the token mapping from the pre-defined export file of the package being converted. The converter will look for the export file in the exportpath.	No
Outputdirectory	Sets the output directory where the output files will be placed.	No
Verbose	If enabled, enables verbose converter output.	No
noWarn	If enabled instructs the Converter to not report warning messages.	No
Mask	If enabled tells the Converter that this package is for mask, so restrictions on native methods are relaxed.	No
NoVerify	If enabled tells the Converter to turn off verification. Verification is turned on by default.	No

TABLE B-5 Parameters for Converter

Attribute	Description	Required
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to <code>true</code> if you do not want the Capgen banner showing.	No
-sign	sign the output CAP file	No
-keystore <keystore>	keystore to use in signing	No
-storepass <storepass>	keystore password	No
-alias <alias>	keystore alias to use in signing	No
-passkey <passkey>	alias password	No
version	Prints Converter version number.	No

Parameters Specified As Nested Elements

AppletNameAID

Use nested element `AppletNameAID` to specify names and AIDs of applets belonging to the package being converted. For details regarding `AppletNameAID` type, see [“AppletNameAID” on page 204](#).

Errors

Execution of this task fails if any of the required elements are not supplied, if `converter.jar` or `offcardverifier.jar` are not in the classpath, or if Converter returns an error code.

Examples

Run Converter to generate helloworld.cap, helloworld.JCA and helloworld.EXP files.

```
<target name="convert_HelloWorld.cap" >
  <convert
    JCA="true"
    EXP="true"
    CAP="true"
    packagename="com.sun.javacard.samples.HelloWorld"
    packageaid="0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1"
    majorminorversion="1.0"
    classdir="${classroot}"
    outputdirectory="${classroot}">
    <AppletNameAID
      appletname="com.sun.javacard.samples.HelloWorld.HelloWorld"
      aid="0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1:0x1"/>
    <exportpath refid="export"/>
    <classpath refid="classpath"/>
  </convert>
</target>
```

In the following example the converter options are specified in helloworld.cfg file instead of being specified in the target itself. This example also shows how a classpath can be specified for a target and how a directory can be set in which the Java VM is invoked for the converter task.

```
<target name="convert_HelloWorld" >
  <convert
    dir="${samples}"
    Configfile="${samples.configDir}/helloworld.cfg">
    <classpath>
      <pathelement path="${samples}"/>
      <fileset dir="${lib}">
        <include name="**/converter.jar"/>
        <include name="**/offcardverifier.jar"/>
      </fileset>
    </classpath>
  </convert>
</target>
```


DeployCap

This task sends a CAP file to `cref` and hides the complexities of creating a script file, running `cref` and then running `APDUTool` to send the script to `cref`. The resulting EEPROM image is saved in the specified output file. This task automatically checks if installation was successful or not by checking status words returned by `cref`.

TABLE B-6 Parameters for DeployCap

Attribute	Description	Required
CapFile	Fully qualified path and name of the CAP file which is to be sent to <code>cref</code> .	Yes
CrefExe	Fully qualified path and name of <code>cref</code> executable.	Yes
OutEEFile	Output EEPROM file that will contain the EEPROM image after <code>cref</code> finishes execution.	Yes
InEEFile	Input EEPROM file for <code>cref</code> . If specified <code>cref</code> initiates using the EEPROM image stored in this file.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to <code>true</code> if you do not want the tool banner showing.	No

Errors and Return Codes

Execution of this task fails if any of the required elements are not supplied, if `apdutool.jar`, `apduio.jar` and `scriptgen.jar` are not in the classpath, or if `APDUTool`, `Scriptgen` or `cref` fail to execute.

Examples

The following example installs `helloworld.cap` file in `cref`. By default it is checked if the APDU commands were sent correctly. Classpath used in the above example is referenced by the `classpath` refid.

```
<target name="Deploy_Hello_world_CAP" >
  <deploycap
    CAPFile="${samples.output}/helloworld.cap"
    outEEFile="${samples.eeprom}/outEEFile"
    CrefExe="{JAVACARD_HOME}/bin/cref">
```

```
<classpath refid="classpath"/>
</deploycap>
</target>
```

The following example installs `helloworld.cap` file in `cref`, which in this case will be initialized with `EEFile`. The `cref` output EEPROM image will also be saved in the same `EEFile`. By default it is checked if the APDU commands were sent correctly. This example shows that the resulting EEPROM image can be stored in the same EEPROM image file that was used to initialize `cref`.

```
<target name="Deploy_Hello_world_CAP" >
  <deploycap
    CAPFile="${samples.output}/helloworld.cap"
    outEEFile="${samples.eeprom}/EEFile"
    inEEFile="${samples.eeprom}/EEFile"
    CrefExe="{JAVACARD_HOME}/bin/cref">
    <classpath refid="classpath"/>
  </deploycap>
</target>
```

Exp2Text

Run Exp2Text tool to convert the export file of a package to a text file.

TABLE B-7 Parameters for Exp2Text

Attribute	Description	Required
PackageName	Fully qualified name of the package.	Yes
ClassDir	Root directory where the exp2text tool will look for the export file. If no ClassDir is specified, the directory in which the Java VM is invoked is taken as base dir.	No
OutputDir	The root directory for output.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to true if you do not want the Exp2Text banner showing.	No
version	Prints Exp2Text version number.	No

Errors

Execution of this task fails if any of the required elements are not supplied, if `converter.jar` is not in the classpath, or if `Exp2Text` returns an error code.

Examples

Run `Exp2Text` to generate text file from the export file of package `HelloWorld`. This example assumes that `converter.jar` is already in classpath.

```
<target name="Exp2TextTarget" >
  <Exp2Text
    packagename="com.sun.javacard.samples.HelloWorld"
    classdir="${classroot}"
    outputdir="${classroot}">
  </Exp2Text>
</target>
```

Run Exp2Text to generate text file from the export file of package HelloWorld. Classdir and the root outputdir are both assumed to be the directory where the Java VM was invoked. Classpath used in this example is referenced by the classpath refid.

```
<target name="Exp2TextTarget" >
  <Exp2Text
    packagename="com.sun.javacard.samples.HelloWorld">
    <classpath refid="classpath"/>
  </Exp2Text>
</target>
```

Maskgen

Runs Maskgen to generate a mask for `cref`, depending on the generator used (see details below).

TABLE B-8 Parameters for Maskgen

Attribute	Description	Required
Generator	Tells Maskgen for which platform is the mask to be generated. Possible choices are <code>a51</code> , <code>cref</code> , and <code>size</code> . For details see Maskgen documentation in the Chapter 10 .	Yes
ConfigFile	Fully qualified path and name of generator specific configuration file.	No
DebugInfo	If enabled, tells Maskgen to generate location debug information for mask.	No
MemRefSize	Integer value that tells Maskgen what memory reference size to use in the mask. Two possible values for element are 16 and 32. Default value used by Maskgen is 32.	No
OutFile	Fully qualified path and name of the output mask file. If this element is not specified, default file name is <code>a.out</code> which will be generated in the directory where the Java VM is invoked.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to <code>true</code> if you do not want the Capgen banner showing.	No
version	Prints Capgen version number.	No

Parameters Specified As Nested Elements

JCAInputFile

Use nested element `JCAInputFile` to specify names of input JCA files for Maskgen. Input JCA files are required to create a Mask file. The reason a standard `FileSet` to specify JCA file names is not used here is that Maskgen supports input file names that starts with an `"@"` symbol to specify an input file that contains a list of names of input JCA files. A file name that starts with `"@"` is not supported by any of the standard Ant types. See description for `"JCAInputFile"` on [page 205](#) for details.

Errors

Execution of this task fails if any of the required elements are not supplied, if `converter.jar` is not in the classpath or if `Maskgen` returns an error code.

Examples

Run `Maskgen` to generate `mask.c` file from input JCA files specified in files `mask1.in` and `mask2.in`.

```
<target name="MasgenTarget" >
  <maskgen
    generator="cref"
    configfile="${maskDir}/mask.cfg"
    outfile="${crefDir}/common/mask.c" >
    <jcainputfile inputfile="@${maskDir}/mask1.in" / >
    <jcainputfile inputfile="@${maskDir}/mask2.in" / >
  </maskgen >
</target >
```

Run `Maskgen` to generate `mask.c` file from input JCA files specified in files `api.in` and `installer` and `helloworld` JCA files.

```
<target name="MasgenTarget" >
  <maskgen
    generator="cref"
    configfile="${maskDir}/mask.cfg"
    outfile="${crefDir}/common/mask.c" >
    <jcainputfile inputfile="@${maskDir}/api.in" / >
    <jcainputfile inputfile="${jcaDir}/installer.jca" / >
    <jcainputfile inputfile="${jcaDir}/helloworld.jca" / >
  </maskgen >
</target >
```

The following example is the same as the previous example, except no output file is specified and classpath is specified. `Maskgen` will generate the file `a.out` in the directory in which Java VM was invoked.

```
<target name="MasgenTarget" >
  <maskgen
    generator="cref"
    configfile="${maskDir}/mask.cfg"
    outfile="${crefDir}/common/mask.c" >
    <jcainputfile inputfile="@${maskDir}/api.in" / >
    <jcainputfile inputfile="${jcaDir}/installer.jca" / >
    <jcainputfile inputfile="${jcaDir}/helloworld.jca" / >
  </maskgen >
</target >
```

```
        <classpath refid="classpath"/>
    </maskgen >
</target >
```

Scriptgen

Runs Scriptgen to generate an APDU script file from a CAP file.

TABLE B-9 Parameters for Scriptgen

Attribute	Description	Required
CapFile	Fully qualified path and name of the input CAP file.	Yes
OutFile	Fully qualified path and name of the output script file. If no output file name is specified, generated script will be output on the console.	No
PkgName	Fully qualified name of the package inside the CAP file.	No
NoBeginEnd	If enabled, instructs Scriptgen to suppress "CAP_BEGIN", "CAP_END" APDU commands.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to true if you do not want the Scriptgen banner showing.	No
version	Prints Scriptgen version number.	No

Errors

Execution of this task fails if any of the required elements are not supplied, if `scriptgen.jar` is not in the classpath or if Scriptgen returns an error code.

Examples

Run Scriptgen to generate script file `helloWorld.scr` from `helloWorld.cap` file. Classpath used in this example is referenced by the `classpath` refid.

```
<target name="ScriptgenTarget" >
  <scriptgen
    noBeginEnd="true"
    noBanner="true"
    CapFile="${samples.helloworld.output}/HelloWorld.cap"
    outFile="${samples.helloworld.script}/helloWorld.scr" >
    <classpath refid="classpath" />
  </scriptgen >
</target >
```


VerifyCap

Runs off-card Java Card platform CAP file verifier to verify a CAP file. The Java Card platform off-card verifier is invoked in a separate instance of Java VM.

TABLE B-10 Parameters for VerifyCap

Attribute	Description	Required
CapFile	Fully qualified path and name of CAP file that is to be verified.	Yes
PkgName	Fully qualified Name of the package inside the CAP file for which the CAP file was generated.	No
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose converter output.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to <code>true</code> if you want to suppress Verifier banner.	No
version	Prints the version number of the off-card verifier.	No

Parameters Specified As Nested Elements

ExportFiles

Use nested element `ExportFiles` to specify group of export files for packages imported by the package whose CAP file is being verified and the export file corresponding to the CAP being verified. For details regarding `ExportFiles` type see [“ExportFiles” on page 205](#).

Errors

Execution of this task fails if any of the required elements are not supplied, if `offcardverifier.jar` is not in the classpath, or if Verifier returns an error code.

Examples

Run the Java Card platform off-card verifier to verify HelloWorld.cap file.

```
<target name="VerifyCapTarget" >
  <verifycap
    CapFile="${samples.helloworld.output}/HelloWorld.cap" >
    <exportfiles file="${samples.helloworld.output}/HelloWorld.exp" />
    <exportfiles file="${api_exports}/javacard/framework/javacard/framework.exp" />
    <exportfiles file="${api_exports}/java/lang/javacard/lang.exp" />
    <classpath refid="classpath"/>
  </verifycap>
</target>
```

VerifyExp

Runs off-card Java Card platform EXP file verifier to verify an EXP file. Java Card platform off-card verifier is invoked in a separate instance of Java VM.

TABLE B-11 Parameters for VerifyExp

Attribute	Description	Required
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose converter output.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to <code>true</code> if you want to suppress Verifier banner.	No
version	Prints the version number of off-card verifier.	No

Parameters Specified As Nested Elements

ExportFiles

Use nested element `ExportFiles` to specify the EXP file being verified. For details regarding `ExportFiles` type see [“ExportFiles” on page 205](#). `VerifyExp` requires that only one input EXP file be specified. This task throws an error if more than one EXP files are specified.

Errors

Execution of this task fails if no EXP file is specified or if more than one EXP file is specified, if `offcardverifier.jar` is not in the classpath, or if Verifier returns an error code.

Examples

Run the Java Card platform off-card verifier to verify HelloWorld.exp file.

```
<target name="VerifyExpTarget" >
  <verifyExp
    <exportfiles file="${samples.helloworld.output}/HelloWorld.exp" />
    <classpath refid="classpath"/>
  </verifyExp>
</target>
```

VerifyRev

Runs off-card Java Card platform verifier to verify binary compatibility between two versions of an EXP file. Java Card platform off-card verifier is invoked in a separate instance of Java VM.

TABLE B-12 Parameters for VerifyRev

Attribute	Description	Required
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose converter output.	No
classpath	Classpath to use for this task. If required jar files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to <code>true</code> if you want to suppress Verifier banner.	No
version	Prints the version number of off-card verifier.	No

Parameters Specified As Nested Elements

ExportFiles

Use nested element `ExportFiles` to specify the EXP files being verified. For details regarding `ExportFiles` type see [“ExportFiles” on page 205](#). `VerifyExp` requires that exactly two input EXP files are specified. This task throws an error if more or less than two EXP files are specified.

Errors

Execution of this task fails if no EXP file is specified or if less or more than two EXP files are specified, if `offcardverifier.jar` is not in the classpath, or if Verifier returns an error code.

Examples

Run the Java Card platform off-card verifier to verify binary compatibility between two versions of HelloWorld.exp file.

```
<target name="VerifyExpTarget" >
  <verifyExp
    <exportfiles file="${samples.helloworld.output}/HelloWorld.exp" />
    <exportfiles file="${samples.helloworld.output.new}/HelloWorld.exp" />
    <classpath refid="classpath"/>
  </verifyExp>
</target>
```

Custom Types

AppletNameAID

AppletNameAID groups together name and AID for a Java Card applet.

TABLE B-13 Parameters for AppletNameAID

Attribute	Description	Required
appletname	Fully qualified name of the Java Card applet.	Yes
aid	AID (Application Identifier) of the Java Card applet.	Yes

Example

Set the fully qualified name and AID for the HelloWorld applet.

```
<AppletNameAID
  appletname="com.sun.javacard.samples.HelloWorld.HelloWorld"
  aid="0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1:0x1" />
```

JCAInputFile

This type is a simple wrapper for a fully qualified JCA file name or a name of an input file that contains a list of input JCA files. In case the input file contains a list of input JCA files, the name of the file should be prepended with “@”.

TABLE B-14 Parameters for JCAInputFile

Attribute	Description	Required
inputfile	Fully qualified name of the input file	Yes

Examples

Set the fully qualified name of an input JCA file.

```
<jcainputfile
  inputfile="C:\jcas\common\com\sun\javacard\installer
\javacard\installer.jca" />
```

Set the fully qualified name of an input file that contains a list of JCA files.

```
<jcainputfile inputfile="@C:\jc\mathDemo.in" />
```

ExportFiles

This type is actually the Ant FileSet type. It is used to specify a group of export files for the off-card verifier. For details, see Apache Ant documentation for FileSet type.

Examples

The following example sets the fully qualified name of an input EXP file.

```
<exportfiles
  file="C:\samples\classes\com\sun\javacard\samples
\HelloWorld\javacard\HelloWorld.exp"
```

The following example groups all the files in the directory `${server.src}` that are EXP files and do not have the text `Test` in their names.

```
<exportfiles dir="${server.src}">
```

```
<include name="**/*.exp"/>
<exclude name="**/*Test*" />
</exportfiles>
```

NetBeans Software Integration

The NetBeans IDE can be used to create an Ant script for a Java technology project and run it to compile, build and run the project. The `build.xml` file for a NetBeans project is located in the project's root directory. You modify the appropriate `build.xml` file as explained in [“Installing the Ant Tasks” on page 179](#) to enable usage of the Ant tasks.

The NetBeans IDE recognizes some pre-defined targets in the `build.xml` file that is generated for a related project. Some of these targets exist, such as `clean` and `compile`, while XML for others is left empty. One of these targets is `-post-compile`. You can modify this target to incorporate usage of the Ant tasks in your project.

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	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 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 <i>-f</i> (or <i>-force</i>) 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 .war extension and is often referred to as a WAR file. A new extension is used instead of .jar because that extension is reserved for files which contain a set of class files and that can be placed in the classpath. As the contents of a web application archive are not suitable for such use, a new extension was required. See <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), 207

AID for installer applet, 87

Ant tasks, 9, 179

 APDUTool, 183

 CapDump, 185

 Capgen, 186

 Converter, 188

 DeployCap, 191

 Exp2Text, 193

 list of, 182

 Maskgen, 195

 Scriptgen, 198

 setting up, 180

 VerifyCap, 199

 VerifyExp, 201

 VerifyRev, 203

APDU

 responses to applet deletion requests, 101

 responses to applet installation requests, 93

 sample script, 96

APDU commands

 sending and receiving, 82

APDU I/O, 139, 143

APDU requests

 to delete applets, 101

 to delete packages, 100

 to delete packages and applets, 100

APDU script file, 183

APDU types, 89

 Abort, 92

 CAP Begin, 90

 CAP End, 91

 Component ## Begin, 91

 Component ## Data, 91

 Component ## End, 91

 Create Applet, 92

 Response, 90

 Select, 90

APDU-based application environment, 207

ApduIOCardAccessor, 142

APDUTool, 183

 errors, 183

 examples of, 184

 parameters, 183

APDUTool task, 183

apdutool tool

 APDU script files, 84

 command line options, 82

 command line syntax, 82

 described, 82

 supported script file commands, 85

API, 208

applet, 208

applet AIDs, 189

applet application, 208

applet container, 208

applet framework, 208

applet instance

 how to create, 88

AppletNameAid element, 189

applets

 APDU responses to deletion requests, 101

 APDU responses to installation requests, 93

- deleting, 99
- application descriptor, 208
- application developer, 208
- application group, 208
- application URI, 208

B

- binary compatibility
 - verifying, 126

C

CAP file

- converting to text, 73, 77
- described, 57
- generating from a Java Card Assembly file, 73, 76
- generating the debug component, 58
- suppressing output, 64
- verifycap tool, 123
- verifying, 123
- versions created, 57

- CAP file production, data flow, 10

CAP files

- how to download, 87
- manifest file example, 75
- manifest file syntax, 73

CapDump, 185

- errors, 185
- examples of, 185
- parameters, 185

capdump tool, 73, 77

- command line syntax, 77

CapFile element, 185

Capgen, 186

- errors, 186
- examples of, 186
- parameters, 186

capgen tool, 73, 76

- command line options, 76
- command line syntax, 76

CardAccessor, 141

classic applet, 208

Classic Edition, 208

clientlib package, 139

com.sun.javacard.javacard.clientlib, 141

com.sun.javacard.javacard.rmiclientlib, 140

command configuration file, 63

Connected Edition, 208

contactless, 45

Converter, 188, 209

- AppletNameAID parameter, 189
 - described, 57
 - errors, 189
 - examples of, 190
 - nested parameters, 189
 - output, 57
 - parameters, 188

converter tool

- command configuration file, 63
- command line options, 60
- command line syntax, 59
- creating a debug.msk file, 65
- input file naming conventions, 63
- invoking the off-card verifier, 64
- Java Card Assembly syntax example, 153
- Java compiler options, 58
- output file naming conventions, 64
- running, 59

converting

- Java class files, 57

cryptography

- support for, 131
- supported keys and algorithms, 131

cryptography classes

- algorithms used by, 133
- instantiating, 134
- supported classes, 132

D

data flow

- installer, 80

debug component

- generating in the CAP file, 58

debug.msk file

- creating, 65

deletion requests

- how to send, 99

demonstrations

- biometric demo, 45
- cryptography demo, 51, 52
- Java Card RMI demo, 38
- logical channels demo, 26
- PhotoCard demo, 36

- Secure Java Card RMI demo, 39
 - transit demo, 53
- DeployCap, 191
 - errors, 191
 - examples of, 191
 - parameters, 191
- description
 - samples, 21
- Development Kit
 - Normalizer tool, 69
 - uninstalling, 19
- distribution format, 209

E

- EEPROM, 105
- EEPROM image files, 111
- Exp2Text, 193
 - errors, 193
 - examples of, 193
 - parameters, 193
- exp2text tool, 67
- export file
 - converting to text, 67
 - loading, 65
 - verifying, 123, 125
- export map
 - specifying, 66
- ExportFiles
 - examples of, 205
- extended applet, 209

I

- input file
 - naming conventions for the converter tool, 63
- input files
 - suppressing verification, 64
 - verifying, 64
- input files for the RI, 111
- installation
 - Java Communications API, 12
- installer
 - components, 80
 - data flow, 80
 - described, 79
 - limitations, 103
- installer applet AID, 87

J

- Java Card Assembly file
 - syntax example, 153
 - using to generate a CAP file, 73, 76
- Java Card RE
 - contents of an implementation, 4
- Java Card RI. *See* RI
- Java Card RMI client
 - reference implementation, 139
 - remote stub object, 140
 - supported framework package, 139
 - supported reference implementation package, 139
- Java Card TCK, 5
- Java Communications API
 - installing, 12
- Java compiler options
 - setting for the converter tool, 58
- JCA files, 195
- JCCardObjectFactory, 141
- JCCardProxyFactory, 141
- JCRemoteRefImpl, 141
- JCRMIConnect, 141

L

- library dependencies, 181

M

- Maskgen, 195
 - errors, 196
 - examples of, 196
 - JCAInputFile parameter, 195
 - parameters, 195

N

- NetBeans
 - modified build.xml
 - Netbeans, modifying for, 206
 - software integration, 206
- Normalizer tool, 69
- normalizer.bat, 69

O

- off-card verifier, 123
 - invoking, 64
 - suppressing verification, 64

- output file
 - naming conventions for the Converter tool, 64
- output files
 - for the RI, 111
 - suppressing verification, 64
 - verifying, 64

P

- packages
 - deleting, 99
- protected content, 211

R

- reimplementing a package or method, 66
- remote stub object, 140
- RI, 105
 - command line syntax and options, 106
 - EEPROM image files, 111
 - features supported, 105
 - input and output, 111
 - limitations, 110
 - running, 106
- RMIC compiler, 140
- rmiclientlib package, 139
- ROM mask, 113

S

- samples
 - building, 21
 - Channels sample, 27
 - description, 21
 - HelloWorld sample, 25
 - ObjectDeletion sample, 32
 - running, 22
- Scriptgen, 198
 - errors, 198
 - examples of, 198
 - parameters, 198
- scriptgen tool
 - command line options, 81
 - command line syntax, 81
 - described, 81
- store files, 111
- stub object, remote, 140

T

- TCK *see* Java Card TCK
- Technology Compatibility Kit *see* Java Card TCK
- thread's active context, 213
- TLV, 29

U

- uninstalling the Development Kit, 19

V

- VerifyCap, 199
 - errors, 199
 - examples of, 200
 - ExportFiles parameter, 199
 - parameters, 199
- verifycap tool, 123, 124
 - command line options, 127
 - command line syntax, 124
- VerifyExp, 201
 - errors, 201
 - examples of, 202
 - ExportFile parameter, 201
 - parameters, 201
- verifyexp tool, 125
 - command line options, 127
 - command line syntax, 125
- VerifyRev, 203
 - errors, 203
 - examples of, 204
 - ExportFiles parameter, 203
 - parameters, 203
- verifyrev tool, 126, 127
 - command line options, 127
 - command line syntax, 127