Java Card

Development Kit User Guide

Version 3.1.0u5

E99052-07

March 2021

Java Card Development Kit User Guide, Version 3.1.0u5

E99052-07

Contents

Preface

Part I Setup, Samples and Tools

1 Introduction

2 Installation

3 Developing Java Card Applications

4 Running the Samples

5 Converting and Exporting Java Class Files

Part II Programming With the Development Kit

11 Using Object, CAP File, and Applet Deletion

12 Working with Logical Channels

13 Using Java Card RMI

14 Using Extended APDU

15 Working with APDU I/O

16 Programming for the Large Address Space

17 Programming Large Java Card Applications With Multiple Packages

18 Java Card Accessibility Information

Part III Java Card Eclipse Plug-in

19 Using the Java Card Eclipse Plug-in

Glossary

List of Tables

Preface

This document describes how to use the Java Card Development Kit, Version 3.1.0u5 to develop Java Card applets.

Java Card technology combines a subset of the Java programming language with a runtime environment optimized for secure elements, such as smart cards and other tamper-resistant security chips. Java Card technology offers a secure and interoperable execution platform that can store and update multiple applications on a single resource-constrained device, while retaining the highest certification levels and compatibility with standards. Java Card developers can build, test, and deploy applications and services rapidly and securely. This accelerated process reduces development costs, increases product differentiation, and enhances value to customers.

The Java Card API is compatible with international standards for secure elements, such as ISO 7816 or mobile communication standards issued by ETSI/3GPP. Major industry-specific standards, such as EMVCo and Global Platform refer to this standard.

> **Note:**
>
> The Java Card Development Kit, Version 3.1.0u5 is released in both binary and source bundles. The binary bundles are the Java Card Development Kit Simulator and the Java Card Development Kit Tools bundles publicly available on OTN. Access to the source bundles requires the purchase of a commercial license from Oracle. Besides, some source bundles may not include cryptography extensions, which are subjected to export restrictions. A few portions of this document are targeted toward source bundles with or without cryptography extensions and are identified as such throughout this book.

Audience

This *Development Kit User Guide* is written for developers who are creating applets using the *Application Programming Interface, Java Card Platform, Version 3.1* 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, you should be familiar with the Java programming language and secure element technology.

You should also become familiar with the Java Card specifications, which are located at Java Card Documentation.

Information on Java Card technology, including access to the latest Java Card Development Kit downloads, is available at `https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html`.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

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

- *Java Card Platform Application Programming Interface Specification, Classic Edition, Version 3.1*
- *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1*
- *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.1*
- *Off-Card Verifier for the Java Card Platform White Paper*
- *Java Card RMI Client Application Programming Interface* (see the Javadoc tool generated API specification at *JC_HOME_SIMULATOR*`\docs\rmiclientlib`)
- *ISO 7816-4:2013 Specification*

Documentation and Support

These web sites provide additional resources:

- Java Card Documentation
- Support `https://www.oracle.com/us/support`

Third-Party Web Sites

Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle 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. Oracle 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.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

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. It contains the following chapters:

- Introduction
- Installation
- Developing Classic Edition Applications
- Running the Samples
- Converting and Exporting Java Class Files
- Working With CAP Files
- Debugging Applications
- Packaging and Deploying Your Application
- Verifying CAP and Export Files
- Using Cryptography Extensions

1
Introduction

The Java Card Platform consists of two editions, the Classic Edition and the Connected Edition. This document and development kit apply only to the Classic Edition. Refer to the Java Card Platform Version 3.0.5 documentation for information on Java Card Connected Edition.

The Java Card Development Kit is a suite of tools for designing implementations of Java Card technology and developing applets based on the Java Card API Specification. It is available as two independent downloads:

- The Java Card Development Kit Tools are used to convert and verify Java Card applications. The Tools can be used with products based on version 3.1 of the Java Card specifications, and should also be used with products based on versions 3.0.4 and 3.0.5 of the Java Card Platform specifications, Classic Edition.
- The Java Card Development Kit Simulator offers a testing and debugging reference for Java Card applications. It includes a Java Card simulation environment and Eclipse plug-in. It provides support for the latest Java Card 3.1 Specification, and can also run applications written for earlier releases.

Together, these two downloads provide a complete, stand-alone development environment in which applications written for the Java Card platform can be developed and tested.

For detailed information on bundles content, refer to the *Java Card Development Kit Tools Release Notes* and *Java Card Development Kit Simulator Release Notes*. The Java Card Development Kit Simulator is only designed as an example of the functional behavior of a Java Card runtime. It is not intended to operate in a production environment (and under the threats typically associated with such an environment).

This chapter contains the following sections:

- Java Card Platform Architecture
- Java Card TCK

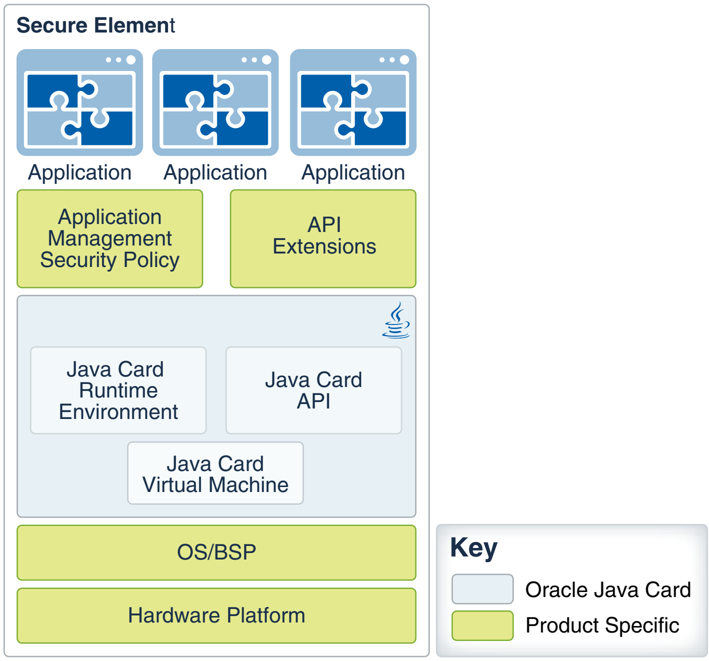
Java Card Platform Architecture

Any implementation of a Java Card Runtime Environment (Java Card RE) contains a Virtual Machine (VM) for the Java Card platform and the Java Card Application Programming Interface (API) classes.

The Java Card Platform, Classic Edition is targeting resource-constrained devices that solely support applet-based applications. Applets that run on the Classic Edition are sometimes referred to as classic applets.

The Java Card Platform, Version 3.1 architecture illustrated below is built on the Classic Java Card VM. It preserves backward compatibility with Classic Applets written for earlier versions.

Figure 1-1 Classic Edition Architecture



This development kit includes a Reference Implementation of the Java Card RE, which stands for the simulator throughout this book. It is invoked on the command line with `cref.bat`. It implements the ISO 7816-4:2013 specification, including support for up to 20 logical channels and the extended APDU extensions as defined in ISO 7816-3.

Java Card TCK

The Java Card Technology Compatibility Kit (Java Card TCK) is a configurable automated test suite for verifying the compliance of an implementation with the applicable Java Card specification. To be in compliance, an implementation must pass the Java Card TCK tests as described in the *Java Card Technology Compatibility Kit User Guide, Version 3.1.0u4*.

The Java Card TCK is available to developers who are considering creating a vendor-specific framework based on the Java Card specifications, under a commercial license from Oracle.

2 Installation

This chapter describes the software that you must install on your system before you can use the development kit, how to install the development kit, how to check system variables, and how to uninstall the development kit.
This Section only applies to the binary bundles of the Java Card Development Kit Simulator and the Java Card Development Kit Tools.

The development kit is available as two independent downloads:

- The Java Card Development Kit Tools are used to convert and verify Java Card applications. The Tools can be used with products based on versions 3.1, 3.0.5, and 3.0.4 of the Java Card Specifications.

> **Note:**
>
> Using Tools with products based on versions 3.0.5 and 3.0.4 applies to Classic scope only .

- The Java Card Development Kit Simulator offers a testing and debugging reference for Java Card applications. It includes a Java Card simulation environment and Eclipse plug-in. It provides support for the latest Java Card 3.1 Specification, and can also run applications written for earlier releases.

This chapter contains the following sections:

Install and Setup the Java Card Development Kit Simulator

The Java Card Development Kit Simulator offers a testing and debugging reference for Java Card applications. It includes a Java Card simulation environment and Eclipse plug-in.

It provides support for the latest Java Card 3.1 Specification, and can also run applications written for earlier releases.

Although the Simulator bundle includes the Java Card Development Kit Tools for convenience of installation, it is recommended that you download and install the latest version of the Java Card Development Kit Tools separately to ensure benefits from the latest security fixes.

This section describes how to install and set up the Java Card Development Kit Simulator. It includes procedures for performing the following tasks:

- Before Installing the Development Kit Simulator
- Installing the Development Kit Simulator
- Confirming System Variables

Before Installing the Java Card Development Kit Simulator

Before installing the Java Card Development Kit Simulator, be sure to install the following software:

- **Java Development Kit (JDK)** - The tools in this development kit were tested with Oracle JDK 11 (64 bit version) and OpenJDK 11 (64 bit version). If you are planning to develop your own applications, you should use JDK 11. You can download and install the JDK release according to the instructions on the website:

  `http://www.oracle.com/technetwork/java/javase/downloads`

- **Eclipse IDE (optional)** - Using the Eclipse IDE as your development environment is recommended, although you can also run the samples and the development kit tools from the command line.

  Download the Windows Eclipse IDE (Eclipse Neon, Oxygen, or Photon) from the following URL, and install it according to instructions on the website:

  `http://eclipse.org/downloads`

- **Apache Ant** - Most Eclipse distributions include Apache Ant. If you did not install Eclipse, you should install Apache Ant, as it is required to run the samples from command line and to build the `cref` from source code. Version 1.9.13 was used to test the release. You can download and install Apache Ant from `http://ant.apache.org`.

- **GCC compiler (For source package only)** - Minimal GNU for Windows (MinGW), version 6.3.0 is required to build the `cref` and tools from sources.

  You can download MinGW from `http://sourceforge.net/projects/mingw`. For MinGW installation information, go to `http://www.mingw.org`.

Installing the Java Card Development Kit Simulator

Follow these steps to install the Java Card Development Kit Simulator.

The Java Card Development Kit Simulator is available for download at `https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html`.

1. Close Eclipse, if it is running.
2. Download the Java Card Development Kit Simulator `.msi` file to a directory of your choice.
   - `java_card_simulator-3_1_0-ux-win-bin-`*`do|gl-buildID-dd_mmm_yyyy`*`.msi`
3. Run the downloaded `.msi` file from the directory.
   a. The Java Card Development Kit Simulator Setup Wizard starts. Follow the prompts and accept the License Agreement.

b. Enter a directory where the files will be installed and follow the prompts to complete the process.

> **Note:**
>
> The installation directory is referred to as *`JC_HOME_SIMULATOR`* throughout the documentation.

When the Java Card Development Kit Simulator has been installed, proceed to:

1. Optional, but recommended. Install the Java Card plug-in for Eclipse. See Installing the Eclipse Plug-In
2. Examine and run the samples. See Running the Samples

Confirming System Variables

Certain system variables are set during the installation process. If you are not able to build samples from the command line, or if something seems to be wrong with the Eclipse plug-in operation, verify that the following variables and paths are set correctly:

- *JAVA_HOME* system variable must be set to the JDK software root directory and its `bin\` in the `PATH`.
- *ANT_HOME* system variable must be set to the Ant root directory and its `bin\` in the `PATH`.
- *JC_HOME_SIMULATOR* variable must be set to the Java Card Development Kit Simulator root directory.
- The Java Card Development Kit `bin\` directory must be in the `PATH` .
- The MinGW `bin\` directory must be in the `PATH`. MinGW is only required if the development kit source bundle is installed.

Install and Setup the Java Card Development Kit Tools

The Java Card Development Kit Tools are used to convert and verify Java Card applications. The Tools can be used with products based on version 3.1, 3.0.5, and 3.0.4 of the Java Card Specification.

The Java Card Development Kit Tools can be used stand-alone, or in conjunction with the Java Card Development Kit Simulator.

This section describes how to install and set up the Java Card Development Kit Tools. It includes procedures for performing the following tasks:

- Before Installing the Java Card Development Kit Tools
- Installing the Java Card Development Kit Tools
- Confirming System Variables

Before Installing the Java Card Development Kit Tools

Before installing the Java Card Development Kit Tools, make sure to install the following software:

Java Development Kit (JDK) - The tools in this development kit were tested with Oracle JDK 11 (64 bit version) and OpenJDK 11 (64 bit version). You can download and install the JDK release according to the instructions on the website:

`http://www.oracle.com/technetwork/java/javase/downloads`

Installing the Java Card Development Kit Tools

Follow these steps to install the Java Card Development Kit Tools.

The Java Card Development Kit Tools is available for download at `https://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html`.

1. Download the Java Card Development Kit Tools `.zip` file to a directory of your choice.
   - `java_card_tools-<ea>-win-bin-`*`buildID-<dd_mmm_yyyy>`*`.zip`
2. Extract the downloaded `.zip` file to the directory of your choice.

> **Note:**
>
> The installation directory of the Java Card Development Kit Tools is referred to as `JC_HOME_TOOLS` throughout this documentation

Confirming System Variables

Certain system variables are set during the installation process. If you are not able to build samples from the command line, or if something seems to be wrong with the Eclipse plug-in operation, verify that the following variables and paths are set correctly:

- `JAVA_HOME` system variable must be set to the JDK software root directory and its `bin\` in the `PATH`.
- `ANT_HOME` system variable must be set to the Ant root directory and its `bin\` in the `PATH`.
- *JC_HOME_TOOLS* variable must be set to the Java Card Development Kit Tools root directory.
- The Java Card development kit `bin\` directory must be in the `PATH` .
- The MinGW `bin\`directory must be in the `PATH`. MinGW is only required if the Development Kit source bundle is installed.

Installed Files and Directories

If you have installed only the Java Card Development Kit Tools bundle, the installation directory is referred to by the environment variable *JC_HOME_TOOLS* in this guide. All the files and directories contained in the Tools bundle are installed in this directory.

If you have installed the Java Card Development Kit Simulator and the Java Card Development Kit Tools, the installation directory of the Simulator is referred to by the environment variable *JC_HOME_SIMULATOR* and the installation directory of the Tools is referred to by the environment variable *JC_HOME_TOOLS* in this guide.

Eclipse Java Card Plug-in

The Java Card development Kit Simulator includes an Eclipse plug-in to assist you in developing Java Card applications. Almost all of the choices presented by the plug-in dialogs correspond to command-line options of the tools included in the Java Card Development Kit Simulator or Java Card Development Kit Tools bundle (`cref`, `converter`, `scriptgen`, and `apdutool`) which are described elsewhere in this user guide. The plug-in runs those tools with the options that you select.

This section includes procedures for performing the following tasks:

- Installing the Eclipse Plug-In
- Setting Up the Java Card Platform

Installing the Eclipse Plug-in

The Eclipse plug-in provides a convenient way to develop Java Card applets. To install the plug-in:

1. In the Eclipse menu bar, select **Help**, and then select **Install New Software**.
2. Click **Add**.
3. Click **Archive**.
4. Select the development kit's Eclipse plug-in repository file:

   JC_HOME_SIMULATOR`\eclipse-plugin\jcdk-repository_`*yyyymmddxxxx*`.zip`
5. On the Add Repository dialog, type `Java Card SDK` in the **Name** field.
6. Click **Add**.
7. On the Available Software dialog, select the feature to install:

   Java Card 3 Platform Development Kit

   > **Note:**
   >
   > If you don't see any items and the message, `There are no categorized items` appears, then you must uncheck the **Group items by category** checkbox.
8. Click **Next** until the terms of the licenses are displayed.
9. Accept the terms of the licenses and click **Finish** to complete installation.
10. When the software has been installed, Eclipse will prompt you to restart. Click **Yes**.

Continue to Configuring Sample_Platform and Sample_Device

Configuring Sample_Platform and Sample_Device

When you create a Java Card project in Eclipse, you must identify the platform, which is the location of the development kit, and provide settings for the device that the simulator will create. You may have more than one simulated device associated with a platform.

When you start Eclipse with the plug-in installed, the plug-in creates

- Sample_Platform, which points to the directory that is set in the *JC_HOME_SIMULATOR* environment variable, and
- Sample_Device, which contains the settings for cref (the simulator).

Sample_Platform and Sample_Device are used for running samples, but you can use them for your own programs too. If they are not created successfully, create them manually using the instructions below.

To change the directory of the Java Card platform or the device settings for a project:

1. In Eclipse, from the **Window** menu, select **Preferences**.
2. In the Preferences dialog, click on the arrow to the left of **Java Card Platforms**.
3. Now you can select **Java Card Platforms** to add, delete or update platforms, and **Java Card Devices** to add, delete or update the simulated device settings.

Configuring the Java Card Tools Path

To enable the Eclipse plugin to build a Java Card project, you must configure the Java Card tool path. This is the path where the tools bundle is present. The tools bundle includes the `converter` and `verifier` tools. This is a workspace setting. Therefore, the tools bundle is used for all Java Card projects that are created in the same workspace.

To change the Java Card tools path for the workspace:

1. In Eclipse, from the **Window** menu, select **Preferences**.
2. In the Preferences dialog, click the **Java Card Tools Path** section.
3. In the Java Card Tools Path dialog, click **Browse** and select the directory where the Java Card tools bundle is installed.

Uninstalling the Java Card Development Kit Simulator

To uninstall the Java Card Development Kit Simulator, do the following:

1. Start Eclipse and remove the Java Card plug-in:
   a. From the Eclipse **Help** menu, select **About Eclipse**.
   b. On the **Installed Software** tab, select **Java Card 3 Platform Development Kit**.
   c. Click **Uninstall...** and follow the prompts.
2. Remove Windows registry entries by running the uninstaller. From the Windows Control Panel:
   a. Click **Programs and Features**.
   b. Select **Java Card Development Kit Simulator** from the list of programs.
   c. Click **Uninstall** and then **Finish**.
3. Delete the `JC_HOME_SIMULATOR` directory from your hard drive, if required.

Uninstalling the Java Card Development Kit Tools

Perform the following steps, if you have installed the Java Card Development Kit Tool only:

1. Delete the `JC_HOME_TOOLS` directory from your hard drive.
2. Delete the *JC_HOME_TOOLS* environment variable.

If you have installed the Java Card Development Kit Tools with the Java Card Development Kit Simulator, perform the following step:

1. Delete the *JC_HOME_TOOLS* environment variable.

3 Developing Java Card Applications

This chapter provides an introduction to developing Java Card applications. You should also refer to the *Application Programming Interface, Java Card Platform, Classic Edition, Version 3.1* for additional information.
This chapter contains the following sections:

- Java Card Applet Development
- Java Card Development Kit Components
- Using Java Card Development Kit Tools

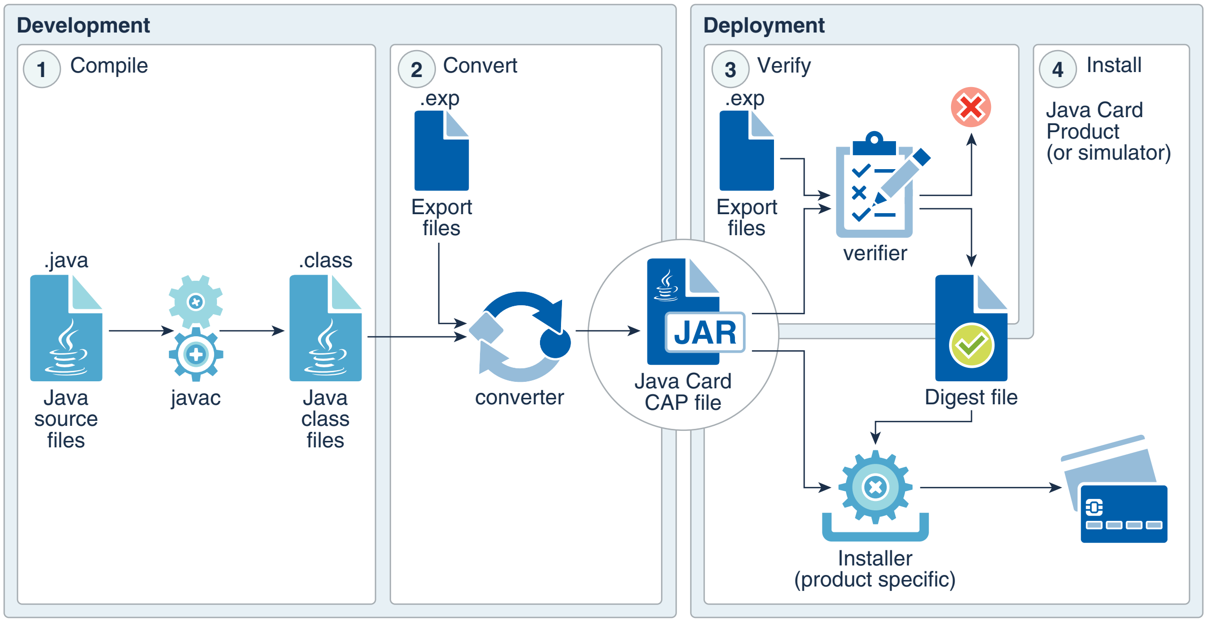
Java Card Applet Development

To develop an applet, you should do the following:

- **Install and Setup** — Install and setup the development environment. Using the Eclipse IDE and Java Card plug-in is recommended. See Installation.
- **Review Samples** — Read Running the Samples, run the samples, and examine the code.
- **Develop** — Develop your applet and compile the code to create the Java class files. Then use the Java Card Development Kit Tools to convert the classes and create a CAP file that can be downloaded to the simulator. See Using the Java Card Development Kit Tools for more information on how to use the tools. Also, see the chapters in Part II for more information about various programming issues.
- **Deploy** — Deploy your application to the Java Card simulator. See Packaging and Deploying Your Application.
- **Debug** — Debug the applet. Use the Java Card debug proxy included in the development kit. See Debugging Applications.

The figure shows the applet development and deployment process.

Figure 3-1 Process for Applet Development and Deployment



Java Card Development Kit Components

The development kit is available as two independent downloads.

Java Card Development Kit Simulator - Includes a Java Card simulation environment, Eclipse plug-in, and the associated testing and debugging tools. It provides support for the latest Java Card Specification, and can also run applications written for earlier releases.

- **cref** - The Java Card simulator. There are three versions of `cref` to handle various communication protocols.
- **scriptgen** - The off-card installer, of which `scriptgen` is a part, resides on the desktop and generates script files for `apdutool`'s use. See Packaging and Deploying Your Application.
- **apdutool** - A client-side tool, which sends the APDU commands to the RE and your on-card applet application. During the application deployment process, you can use it to read the output script file generated by `scriptgen` to send it to the Card Manager application. See Packaging and Deploying Your Application.
- **Eclipse plug-in** - The plug-in provides a way to run the rest of the tools in this list from inside Eclipse. Running the Samples in Eclipse

Java Card Development Kit Tools - Used to convert and verify Java Card applications. The Tools can be used with products based on version 3.1, 3.0.5, and 3.0.4 of the Java Card Specifications.

- **Converter** - Converts Java classes into a CAP file, one or more Java Card Assembly files, or one or more export files. See Converting and Exporting Java Class Files.

- **verifier** - Verifies the contents of a smart card using verifycap, verifyexp, and verifyrev. See Verifying CAP and Export Files.
- **capgen** - Generates a compact CAP file from a Java Card Assembly file, or an extended CAP file from one or more Java Card Assembly files. See Working With CAP Files.
- **capdump** - Creates an ASCII version of a CAP file. See Working With CAP Files.
- **exp2text** - Enables you to view any export file in text format. See Converting and Exporting Java Class Files.
- **ant tasks** - Set of tasks to use the tools in ant scripts. See Setting Up the Optional Ant Tasks.

Using Java Card Development Kit Tools

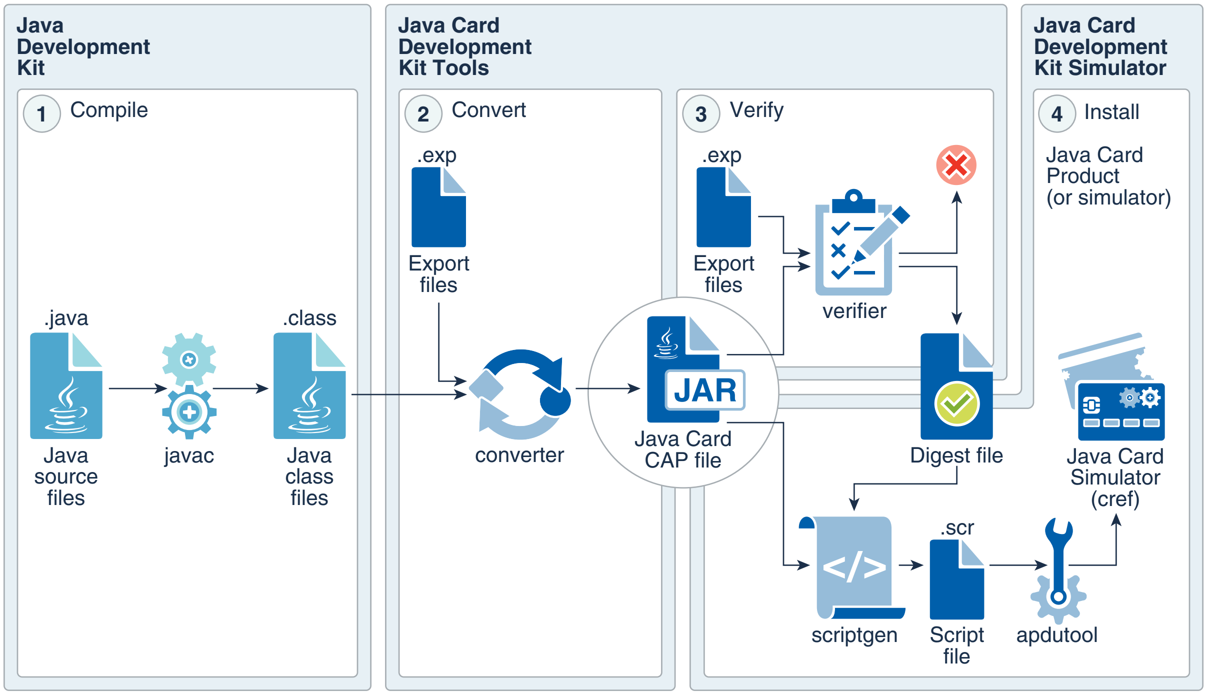
Use the Java Development Kit to compile your applet. Then, use the converter tool from the Java Card Development Kit Tools to convert the compiled Java file (.class file) to a Java Card CAP file.

A Java Card CAP file is a JAR file containing the binary representation of a unit of code, made of one or more Java packages. It can be distributed for deployment on real devices running Java Card or simulators.

The Deployment process consists of verifying the CAP file and installing the code on the device. The verifier tool from the Java Card Development Kit Tools does the verification. The installation depends on the target device and uses the specific installation tools.

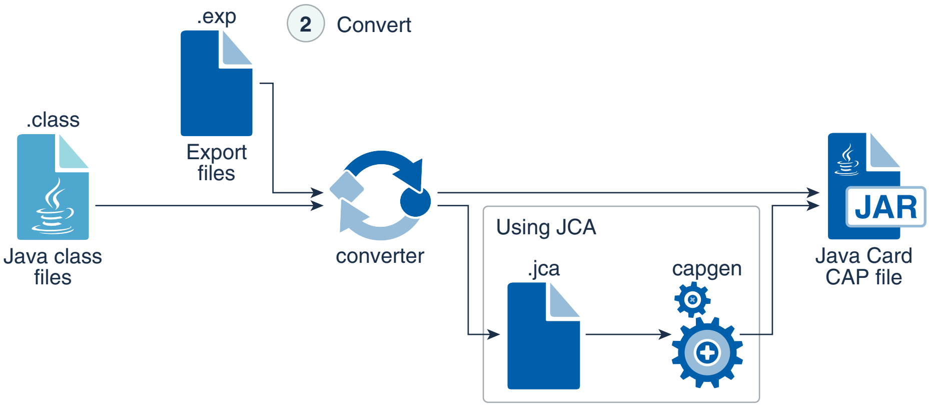
To deploy the CAP files on the Java Card simulator (cref), use the scriptgen tool, which produces an installation script, made of APDU commands, that can then be transmitted to the Java Card simulator using the apdutool tool. The following illustration depicts the Java Card tool chain.

Figure 3-2 Java Card Tool Chain



Note that you can use the Converter tool to produce Java Card Assembly (JCA) files. A JCA file is a textual representation of a converted package that you can use to aid testing and debugging. You can use a JCA file as an input to the `capgen` tool to create a CAP file. The following illustration depicts this process.

Figure 3-3 Using Java Card Assembly



4

Running the Samples

A number of example programs are provided with the development kit. Two directories containing samples are located under `JC_HOME_SIMULATOR\samples`:

- `classic_applets` show basic functionality.
- `reference_apps` are outlines of applications that demonstrate the interactions between various applications on the card using features such as SIO and events.

This chapter contains the following sections:

- How to Run the Samples
- Running the classic_applets Samples
- Running the reference_apps Samples

How to Run the Samples

Each sample directory contains an `applet` folder and, if applicable, `client` folder. You can use the Eclipse plug-in or the `ant` tool, which is invoked from the command line, to build and run the samples. In either case, the outcome is the same: the development kit tools are used to convert the class files and generate APDU script files.

The Java Card runtime environment, `cref`, simulates a Java Card Platform, Version 3.1 on a smart card. Applets are installed in the runtime environment, and it simulates interaction with a card reader.

Included in each sample directory is an expected output file so that you can see if the sample is running correctly.

To build and run the samples, go to one of the following:

- Running the Samples in Eclipse
- Running the Samples from the Command Line.

Running the Samples in Eclipse

To run a sample, you import the project, build the project, start the device, run the CAP script to install the code, and then run the sample-specific script. Detailed instructions are provided for running each of the samples using Eclipse. Some instructions vary in how they do a task, so that you can learn about the plug-in as you follow along.

Almost all of the choices presented by the plug-in dialogs correspond to command-line options of the development kit tools (`cref`, `converter`, `scriptgen` and `apdutool`) which are described elsewhere in this guide. The plug-in runs those tools with the options that you select.

Following are a few notes on running the samples.

Sample_Platform and Sample_Device

When you start the Eclipse with the plug-in installed, it automatically creates or re-creates Sample_Platform and Sample_Device. If for some reason they are not created, refer to the instructions in Configuring Sample_Platform and Sample_Device.

Java Card View

The sample instructions refer to the Java Card view. If you don't see the Java Card view, go to the **Window** menu, select **Reset Perspective...** Click **Yes** to confirm the reset.

Importing and Building Projects

Using the **File** menu, select **Import > General > Projects from Folder or Archive** to import a Java Card project. Make sure that you select the directory that has Java Card source files in it from the project. In most cases, this directory is the `applet` folder.

After you import a project, the build starts (if **Build Automatically** under the **Project** menu is selected) and generates the following artifacts for each Java package:

- `deliverables` — `cap`, `jca`, and `exp` files
- `cap*.script` — for installing the package
- `create*.script` — for installing the applet
- `select*.script` — for selecting the applet

The scripts are put in the `apdu_scripts` directory. The outputs from the converter (`cap`, `jca`, and `exp` files) are put in the the `deliverables` directory.

Running Sample_Device

Start `cref` by right-clicking on Sample_Device in Java Card View and selecting **Start**. The console opens with the output from `cref` and `apdutool`, and a prompt, `CMD>` You can enter an APDU command, which is sent to the card (Sample_Device), and the response is displayed on the console.

One simple way to test if the console is running is to type the echo command at the prompt:

```
echo "test";
```

You should see the APDU response:

```
test
```

To install a built package, right-click the corresponding `cap*.script` file and select **Java Card** and **Execute Script**.

Sample_Device Settings

Change settings for `cref` by double-clicking on Sample_Device in Java Card View to open the Properties for Sample_Device dialog. From the same dialog you can change the debugger and apdutool settings.

If you do set these parameters, you may need to clear them before running the next sample.

Run Configuration

Run Configuration can be used to automate how scripts are run. You can specify whether `cref` shall be started or re-started, and provide a list of scripts to be executed.

Running the Samples from the Command Line

To build and run the samples:

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

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.

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

   Additional actions might include restarting the simulator 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 - A minimal applet utilizing the simplest source code and meta-files that demonstrates the base structure of a Java Card applet that developers can use to develop, deploy, create, execute, delete, and unload a standalone module.
- Channels Sample - Demonstrates the use of logical channels which allows selecting multiple applets at the same time.
- Service Sample - Demonstrates the Java Card service framework of classes and interfaces that enable a Java Card technology-based applet to be designed as an aggregation of service components.
- Utility Sample - Demonstrates the use of the utility APIs in an applet to simulate stock trading and portfolio management.

- Wallet Sample - Demonstrates a simplified cash card application.
- ObjectDeletion Sample - Contains two samples, `odDemo1` and `odDemo2`, that demonstrate applet and package deletion and the object deletion mechanism that removes unreachable objects.
- PhotoCard Sample - Demonstrates how to store images in the large address space that is available in the 32-bit version of the Java Card simulator.
- 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 can decrement, increment, and return the value of an account. See Programming to the Java Card RMI Client-Side API.
- StringHandlingApp Sample - Demonstrates the use of two Java Card Classic libraries that use string annotations to define string constants and two Java Card Classic applets that use those annotations to define their own set of string constants and import string constants from the libraries.
- SecureRMIPurse Sample - Similar to the `RMIPurse` sample, but demonstrates additional security at the transport level. This sample is only included in bundles with cryptography extensions.
- SignatureMessageRecovery Sample - Demonstrates message recovery. This sample is only included in bundles with cryptography extensions.
- ArrayViews Sample - Demonstrates a client application and a server application sharing data using array views.
- CertHandling Sample - Demonstrates the use of static resources and certificate API to parse and verify a certificate.

HelloWorld Sample

The HelloWorld sample demonstrates the base structure of a Java Card applet.

Follow one of these sets of instructions to run this sample:

- Running the HelloWorld Sample in Eclipse
- Running the HelloWorld Sample from the Command Line

Running the HelloWorld Sample in Eclipse

Run the HelloWorld sample using the APDU console.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `applet` directory from the HelloWorld project, to import the HelloWorld Java Card project into your workspace. If the build doesn't start automatically, start it manually.

   The build generates the scripts and puts them in the `apdu_scripts` directory. It puts the outputs from the converter (`cap`, `jca`, and `exp` files) in the `deliverables` directory.

2. If you don’t see the Java Card view, go to the **Window** menu, select **Show View** and **Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.

3. Before you start any script you must change the `PowerDown` parameters for generating the script files. To change the `PowerDown` parameters:
   a. In the **Package Explorer** view, click the `HelloWorld` Java project .
   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the Java Card CAP Files page.
   d. Click **HelloWorld** and select **Edit**.
   e. In the Edit mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Suppress "PowerDown;" APDU command at the end of CAP script** check box.
   h. Click **Finish** and select **Apply** and **Close**.
4. In the Java Card View, right-click on **Sample_Device** and select **Start**.

   The simulator starts and you can see the output in the Console view.
5. In the Sample_Device console toolbar, click on the **Select script** drop-down and execute these scripts:
   - `cap-com.sun.jcclassic.samples.helloworld`
   - `create-com.sun.jcclassic.samples.helloworld.HelloWorld`
   - `helloworld`

   The scripts are submitted to the simulator and you can see the output.

Compare the output in the Console view with the contents of the `HelloWorld.expected.output` file.

Running the HelloWorld Sample from the Command Line

To run the HelloWorld sample:

1. Open a Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\bin` directory.
   b. Start the simulator 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. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.
   b. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\HelloWorld\applet` directory.

c. 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.

Follow one of these sets of instructions to run this sample:

- Running the Channels Sample in Eclipse

- Running the Channels Sample from the Command Line

Running the Channels Sample in Eclipse

We will run the Channels sample without the APDU console.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the Channels Java Card project into your workspace. If the build doesn't start automatically, start it manually.

   The build creates `apdu_scripts` and `deliverables` directories.

2. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:

   a. In the **Combined (input and output) file for EEPROM data** field, type a file name to be used for saving EEPROM between simulator sessions, e.g., `Channels.eeprom`. The file will be automatically created in the bin directory. Later, after the sample run, you can safely delete it.

   b. Select **Do not open APDU console**.

   c. Click **OK**.

3. In the Java Card View, right-click on **Sample_Device** and select **Start**.

   The simulator starts and you can see the output in the Console view.

4. In the Package Explorer window, expand the `apdu_scripts` folder, right-click on `cap-com.sun.jcclassic.samples.channels.script`, and select **Java Card** and **Execute Script**.

   You see the simulator output in the Console view. The simulator is stopped.

5. Right-click on **Sample_Device** and select **Start**.

   The simulator starts and you can see the output in the Console view. This time the simulator restores EEPROM data from the `Channels.eeprom` file saved in the previous session.

6. In the Package Explorer window, in the `apdu_scripts` folder, right-click on `channel.scr`, and select **Java Card** and **Execute Script**.

   You see the simulator output in the Console view and the simulator stopped. Compare the output in the Console view with the contents of the `Channels.expected.output` file.

Running the Channels Sample from the Command Line

To run the Channels sample:

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

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

   b. Start the simulator 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:
   1. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.
   2. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\Channels\applet` directory.
   3. 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.

Follow one of these sets of instructions to run this sample:

- Running the Service Sample in Eclipse
- Running the Service Sample from the Command Line

Running the Service Sample in Eclipse

Run this sample using **Run Configuration** and the APDU console in Eclipse.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `Service` Java Card project into your workspace. If the build doesn't start automatically, start it manually.

The build creates `apdu_scripts` and `deliverables` directories.

2. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:
   1. Clear the **Input file with EEPROM data**, the **Output file for EEPROM data**, and the **Combined (input and output) file for EEPROM data** fields.
   2. Clear **Do not open APDU console**.
   3. Click **OK**.
3. Before you configure, run, and start any script, you must change the `PowerDown` parameters for generating the script files. Otherwise, the simulator goes into the PowerDown mode after running the `cap-Service.script` file and interrupts any execution of the following script files. To change the `PowerDown` parameters:
   1. In the **Package Explorer** view, click `Service` Java project .
   2. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   3. Select a CAP file from the list that appears in the Java Card CAP Files page.
   4. Click **Service** and select **Edit**.
   5. In the Edit mode, select **Compact CAP File**.
   6. Click **Next>**.
   7. Select **ScriptGen** slide and select the **Suppress "PowerDown;" APDU command at the end of CAP script** check box.
   8. Click **Finish** and select **Apply** and **Close**.
4. In the top menu, select **Run** and **Run Configurations...**
5. In the Run Configurations dialog:
   1. Right-click on **Java Card Project Run** and select **New**.
   2. In the **Name** field, enter `Service`
   3. Click **Browse...**, select the Service project, and click **OK**.
   4. Select **Start simulator**.
   5. In the **Scripts to be executed on simulator** list box, add the following scripts:
      - Browse to the *`JC_HOME_SIMULATOR`*`\samples\classic_applets\Service\applet\apdu-scripts` directory and choose `cap-com.sun.jcclassic.samples.service.script`
      - From the same directory, select `service.scr`
   6. Click **Run**

   The simulator starts and executes the scripts in the list box, and you can see the output in the Console view.
   Compare the output with the contents of the `service.expected.output` file.

Running the Service Sample from the Command Line

To run the Service sample:

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

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

b. Start the simulator 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. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.

   b. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\Service\applet` directory.

   c. 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 you can use the utility APIs 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 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 Protection** - PIN protected access to the application. Uses the standard PIN API in the Java Card platform to protect access to the applet.
- **Storage of Portfolio** - Storage of portfolio information on the card. 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 of 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 the user, because of the trade, has 0 stocks of a certain company, the `stockInfo` TLV for that stock is removed from the portfolio constructed TLV.
- Retrieval of complete portfolio information from the card.
- **Get Information On a Stock -** Retrieval of information on a particular stock in the portfolio. 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
- 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.

Follow one of these sets of instructions to run this sample:

- Running the Utility Sample in Eclipse
- Running the Utility Sample from the Command Line

Running the Utility Sample in Eclipse

We will run this sample using **Run Configuration** and the APDU console in Eclipse.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `Utility` Java Card project into your workspace. If the build doesn't start automatically, start it manually.

   The build creates `apdu_scripts` and `deliverables` directories.

2. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:

   a. Clear the **Input file with EEPROM data**, the **Output file for EEPROM data**, and the **Combined (input and output) file for EEPROM data** fields.

   b. Clear **Do not open APDU console**.

   c. Click **OK**.

3. In the top menu, select **Run** and **Run Configurations...**

4. In the Run Configurations dialog:

   a. Right-click on **Java Card Project Run** and select **New**.

   b. In the **Name** field, enter `Utility`

   c. Click **Browse...**, select the Utility project, and click **OK**.

   d. Select **Start simulator**.

   e. In the **Scripts to be executed on simulator** list box, add the following scripts:

      - Browse to the *`JC_HOME_SIMULATOR`*`\samples\classic_applets\Utility\applet\apdu-scripts` directory and choose `cap-com.sun.jcclassic.samples.utility.script`

      - From the same directory, select `UtilityDemoFooter.scr`

   f. Click **Run**

   The simulator starts and executes the scripts in the list box, and you can see the output in the Sample_Device Console view.
   Compare the output with the contents of the `utility.expected.out` file.

Running the Utility Sample from the Command Line

To run the Utility sample:

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

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

   b. Start the simulator 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:
   1. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.
   2. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\Utility\applet` directory.
   3. 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 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.

Follow one of these sets of instructions to run this sample:

- Running the Wallet Sample in Eclipse
- Running the Wallet Sample from the Command Line

Running the Wallet Sample in Eclipse

These instructions use clipboard operations and the APDU console to run the script. You could instead run the script in the usual way (right-click the script, select **Java Card** and **Execute Script**).

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `Wallet` Java Card project into your workspace. If the build doesn't start automatically, start it manually.

   The build creates `apdu_scripts` and `deliverables` directories.

2. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:
    a. Clear the **Input file with EEPROM data**, the **Output file for EEPROM data**, and the **Combined (input and output) file for EEPROM data** fields.
    b. Clear **Do not open APDU console**.
    c. Click **OK**.
3. Before you start any script, you must change the `PowerDown` parameters for generating the script files. Otherwise, the simulator goes into the PowerDown mode after running the `cap-Wallet.script`. To change the `PowerDown` parameters:
    a. In the **Package Explorer** view, click the `Wallet` Java project .
    b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
    c. Select a CAP file from the list that appears in the Java Card CAP Files page.
    d. Click **Wallet** and select **Edit**.
    e. In the Edit mode, select **Compact CAP File**.
    f. Click **Next>**.
    g. Select **ScriptGen** slide and select the **Suppress "PowerDown;" APDU command at the end of CAP script** check box.
    h. Click **Finish** and select **Apply** and **Close**.
4. In Java Card View, right-click on Sample_Device and select **Start**.

    The simulator starts and you can see the output in the Sample_Device console view. The output ends and the **CMD>** prompt is displayed.
5. In the console toolbar, click on the **Select script** drop-down button and select `cap-com.sun.jcclassic.samples.wallet` from the list.

    The script is submitted to the simulator and you can see the output.
6. In Package Explorer, expand the `apdu_scripts` folder and double-click on `wallet.scr` to open it in the editor view. Select all text in the editor view, copy it to the clipboard, and paste it into the Sample_Device console.

    The script is executed by the simulator, and you see the output in the Sample_Device console.

Compare the output with the contents of the `wallet.expected.out` file.

Running the Wallet Sample from the Command Line

To run the Wallet sample:

1. Open a Command Prompt window and perform the following:
    a. Navigate to the `JC_HOME_SIMULATOR\bin` directory.
    b. Start the simulator 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:
   1. Set ANT_HOME (path to ant install folder), JC_HOME_TOOLS and JC_HOME_SIMULATOR (path to JCDK install folder) as environment variables.
   2. Navigate to the JC_HOME_SIMULATOR\samples\classic_applets\Wallet\applet directory.
   3. 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 seven APDU scripts that demonstrate the object deletion mechanism, applet deletion, and package deletion:

- od1-1.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.

  od1-1.scr does not depend on any other sample. The final state of cref memory must be saved to a file for od1-2.scr to use.
- od1-2.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 od1-2.scr sample must be run after od1-1.scr because the initial state of cref must be the same as its final state after running od1-1.scr. After running od1-2.scr, the final state of cref must be saved to a file so that od1-3.scr can use it.
- od1-3.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 od1-3.scr sample must be run after od1-2.scr because the initial state of cref must be the same as its final state after running od1-2.scr.

- od2.scr - Demonstrates package deletion and checks that persistent memory is returned to the memory manager.

  After running od2.scr, the final state of cref must be saved to a file so that od2-2.scr can use it.

- od2-2.scr - Demonstrates shared reference applet deletion and package deletion order.

  The od2-2.scr sample must be run after od2.scr. This is because the initial state of cref must be the same as its final state after running od2.scr.

- od3.scr – Implements a scenario to capture initial memory.

  After running od3.scr, the final state of cref must be saved to a file so that od3-2.scr can use it.

- od3-2.scr - Implements a scenario to verify memory after package deletion.

  The od3-2.scr sample must be run after od3.scr because the initial state of cref must be the same as its final state after running od3.scr.

The simulator must be restarted before running each APDU script.

Follow one of these sets of instructions to run this sample:

- Running the ObjectDeletion Sample in Eclipse
- Running the ObjectDeletion Sample from the Command Line

Running the ObjectDeletion Sample in Eclipse

Run the ObjectDeletion sample without the APDU console.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the ObjectDeletion Java Card project into your workspace. If the build doesn't start automatically, start it manually.

   The build creates apdu_scripts and deliverables directories.

2. In Java Card View, double-click on **Sample_Device**. In the Properties for Sample_Device dialog, select the **CREF** tab:

   a. In the **Combined (input and output) file for EEPROM data** field, type a file name to be used for saving EEPROM between simulator sessions, e.g., ObjectDeletion.eeprom. The file will be automatically created in the bin directory. Later, after the sample run, you can safely delete it.

   b. Select the **Do not open APDU console** check box.

   c. Click **OK**.

3. At the command line, browse to the JC_HOME_SIMULATOR\samples\classic_applets\ObjectDeletion\applet folder.

   a. Set ANT_HOME (path to ant install folder), JC_HOME_TOOLS and JC_HOME_SIMULATOR (path to JCDK install folder) as environment variables.

   b. Build the ObjectDeletion sample and run ant in the JC_HOME_SIMULATOR\samples\classic_applets\ObjectDeletion\applet folder.

A build folder is created with the applet*.scr files.

c. In Eclipse, right-click on the **ObjectDeletion** project and click **Refresh**.

   The build folder is added into the project.

4. Execute the following scripts in the same order from the build folder.

   - applet1-1.scr
   - applet1-2.scr
   - applet1-3.scr
   - applet2.scr
   - applet2-2.scr
   - applet3.scr
   - applet3-2.scr

   a. To run each script, start the simulator, from the Java Card View, right-click on **Sample_Device** and select **Start**.

   b. Right-click on the script file and select **Java Card** and **Execute Script**.

      You see the simulator output in the Console view. The simulator stops after each script run. Compare the output in the Console view with the corresponding expected.out file. Compare the output in the Console view for each run of an applet*.scr file, with the corresponding od*.expected.out file, located in the JC_HOME_SIMULATOR\samples\classic_applets\ObjectDeletion folder.

Running the ObjectDeletion Sample from the Command Line

To run the ObjectDeletion sample:

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

   a. Navigate to the JC_HOME_SIMULATOR\bin directory.

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

      ```
      cref -o e2p
      ```

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

   a. Set ANT_HOME (path to ant install folder), JC_HOME_TOOLS and JC_HOME_SIMULATOR (path to JCDK install folder) as environment variables.

   b. Navigate to the JC_HOME_SIMULATOR\samples\classic_applets\ObjectDeletion\applet directory.

   c. 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 simulator by using ctrl + c.

4. In the cref Command Prompt window, restart the simulator 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 simulator 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 simulator 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 simulator 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 simulator 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 simulator 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 simulator 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 simulator. 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 that communicates with it. The photocard applet 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 and 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 a simulator with different characteristics. The applet code must account for this.

Follow one of these sets of instructions to run this sample:

- Running the PhotoCard Sample in Eclipse
- Running the PhotoCard Sample from the Command Line

Running the PhotoCard Sample in Eclipse

The PhotoCard sample consists of two projects: a Java Card project with the Java Card applet and a Java SE project with the Java application that is designed to communicate with the applet.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the PhotoCard_Applet Java Card project and the PhotoCard_Client Java project into your workspace. You can import both projects in the same Import wizard. If the builds don't start automatically, start them manually.

   The PhotoCard_Applet project build creates apdu_scripts and deliverables directories.

2. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:

   a. In the **Combined (input and output) file for EEPROM data** field, type a file name to be used for saving EEPROM between simulator sessions, e.g., PhotoCard.eeprom. The file will be automatically created in the bin directory. Later, after the sample run, you can safely delete it.

   b. Clear the **Input file with EEPROM data** and the **Combined (input and output) file for EEPROM data** fields.

   c. Clear **Do not open APDU console**.

   d. Click **OK**.

3. In Java Card View, right-click on Sample_Device and select **Start**.

   The simulator starts and you can see that Sample_Device console is created.

4. In Sample_Device console toolbar, click the **Select Script** drop-down button and select cap-com.sun.jcclassic.samples.photocard. Wait until the script execution completes and the **CMD>** prompt is displayed

5. Click the **Select Script** drop-down button again and select `create-com.sun.jcclassic.samples.photocard.PhotoCardApplet`. Verify that the script finished successfully, i.e., with `SW1: 90, SW2: 00`
6. In Sample_Device console toolbar, click the **Stop the device** button.

   The simulator stops, and EEPROM data is saved in `PhotoCard.eeprom` file.
7. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:
   1. In the **Input file with EEPROM data** field, click **Browse..** and select the`PhotoCard.eeprom` file.
   2. Clear the **Output file with EEPROM data**.
   3. Select **Do not open APDU console**.
   4. Click **OK**.
8. In Java Card View, right-click on Sample_Device and select **Start**.

   The simulator starts and you can see the output in the Console view.
9. In the Package Explorer view, right-click on PhotoCard_Client and select **Run As** and **Run Configurations...**
10. In the Run Configurations dialog:
    1. Right-click on **Java Application** and select **New**.
    2. Select the **Arguments** tab.
    3. Enter the following program arguments: `duke_magnify.gif duke_pencil.gif duke_wave.gif duke_thumbsup.gif` and click **Run**

When the program completes you can compare its output with the `photocard-client.expected.out` file.

Running the PhotoCard Sample from the Command Line

To run the PhotoCard sample:

1. Open a Command Prompt window and perform the following:
   1. Navigate to the `JC_HOME_SIMULATOR\bin` directory.
   2. Start the simulator by using the following command at the command prompt:

      ```
      cref -o demoee
      ```

   Starting the simulator with the `-o` option and *filename* causes the simulator to save the EEPROM contents to a file named `demoee`.
2. Open a second Command Prompt window and perform the following:
   1. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.
   2. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\PhotoCard\applet` directory.
   3. 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 simulator by using the following command:

   ```
   cref -z -i demoee
   ```

   Starting the simulator with the `-z` and `-i` options and *filename* causes the simulator to use the contents of the `demoee` file to initialize the EEPROM and to display the resource consumption statistics.
5. In the applet Command Prompt window, perform the following:
   1. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\PhotoCard\client` directory.
   2. 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 an output file (`actual_output.txt`) in the `client` 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 the `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).
>
> Depending on the locale that is used, the presentation of photo sizes might differ between `actual_output.txt` and `photocard-client.expected.out`. For example, "`21,280 bytes`" versus "`21.280 bytes`".

RMIPurse Sample

A Java Card RMI application consists of two parts: a card applet and a client program communicating with it. In this sample, the `RMIPurse` applet is installed in EEPROM image. For further details see Programming to the Java Card RMI Client-Side API.

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.

Follow one of these sets of instructions to run this sample:

- Running the RMIPurse Sample in Eclipse
- Running the RMIPurse Sample from the Command Line

Running the RMIPurse Sample in Eclipse

The RMIPurse sample consists of two projects: a Java Card project with the Java Card applet, and a Java SE project with a Java application that communicates with the applet using RMI.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

This sample uses the `rmic` tool, which is provided with the development kit.

1. Import the `RMIPurse_Applet` Java Card project and the `RMIPurse_Client` Java project into your workspace. You can import both projects in the same Import wizard. If the builds don't start automatically, start them manually.

   The `RMIPurse_Applet` project build creates `apdu_scripts` and `deliverables` directories.

2. Create a launch configuration to specify launch parameters for the `rmic` tool:

   a. From the workbench menu bar, select **Run**, **External Tools**, and **External Tools Configurations...**

   b. On the External Tools Configurations dialog, select **Ant Build** in the left hand list of launch configuration types, and then click **New launch configuration** in the toolbar.

3. On the Main configurations dialog, enter the following text for these fields:

   - **Name:** `rmic`
   - **Buildfile:** `${workspace_loc:/RMIPurse_Applet/build.xml}`

    - **Base directory:** `${workspace_loc:/RMIPurse_Applet}`

    a. On the Targets configurations dialog, select only `rmic` target.

    b. On the Environment configurations dialog, add *JC_HOME_SIMULATOR* and *JC_HOME_TOOLS* variables.

    c. Click **Apply** and **Close**.

4. Import the `RMIPurse_Applet` Java Card project and the `RMIPurse_Client` Java project into your workspace. You can import both projects in the same Import wizard. If the builds don't start automatically, start them manually.

   The `RMIPurse_Applet` project build creates `apdu_scripts` and `deliverables` directories.

5. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:

    a. In the **Output file with EEPROM data** field, type a file name to be used for saving EEPROM between simulator sessions, e.g., `RMIPurse.eeprom`. The file will be automatically created in the `bin` directory. Later, after the sample run, you can safely delete it.

    b. Clear the **Input file with EEPROM data** and the **Combined (input and output) file for EEPROM data** fields.

    c. Clear **Do not open APDU console**.

    d. Click **OK**.

6. In Java Card View, right-click on Sample_Device and select **Start**.

   The simulator starts and you can see that Sample_Device console is created.

7. In the Sample_Device console toolbar, click on the **Select Script** drop-down and execute:

    - `cap-com.sun.jcclassic.samples.rmi` and
    - `create-com.sun.jcclassic.samples.rmi.PurseApplet`

8. Stop the Sample_Device.

9. In the Package Explorer view, expand `RMIPurse_Applet\src\com.sun.jcclassic.samples.rmi` and select `PurseImpl.java`. From the Eclipse menu bar, select **Run**, then **External Tools** and **rmic**. (If this is the first rmic execution, select **Run**, **External Tools** , **External Tools Configurations...** and click **Run**).

   Click **F5** to refresh the view, and you should see `PurseImpl_Stub.class` in the `RMIPurse_Applet\stubs\com\sun\jcclassic\samples\rmi` directory.

10. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:

    a. In the **Input file with EEPROM data** field, click **Browse...** and select the `RMIPurse.eeprom` file.

    b. Clear the **Output file for EEPROM data** field.

    c. Select **Do not open APDU console**.

    d. Click **OK**.

11. In Java Card View, right-click on Sample_Device and select **Start**.

    The simulator starts and you can see the output in the Console view.

12. In the Package Explorer view, expand `RMIPurse_Client` project, navigate to `PurseClient.java`, right-click on it, and select **Run As** and **Java Application**.

You see the application output in the console. Now you can compare it with the contents of `rmidemo.expected.output` file.

Running the RMIPurse Sample from the Command Line

To run the RMIPurse sample:

1. Open a Command Prompt window and perform the following:
   1. Navigate to the `JC_HOME_SIMULATOR\bin` directory.
   2. Start the simulator by typing the following command at the command prompt:

      ```
      cref -o demoee
      ```

   Starting the simulator with the `-o` option and *filename* causes the simulator to save the EEPROM contents to a file named `demoee`.

2. Open a second Command Prompt window and perform the following:
   1. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.
   2. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\RMIPurse\applet` directory.
   3. 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 simulator by using the following command:

   ```
   cref -i demoee
   ```

   Starting the simulator with the `-i` option and *filename* causes the simulator to use the contents of the `demoee` file to initialize the EEPROM.

4. In the applet Command Prompt window, perform the following:
   1. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\RMIPurse\client` directory.
   2. 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 `RMIPurse` directory.

StringHandlingApp Sample

The StringHandlingApp sample demonstrates how annotations and string utility methods for the Java Card platform are used and showcases the use of:

- Annotations in javacardx.annotations.StringDef and javacardx.annotations.StringPool
- String utility methods defined in javacardx.framework.string.StringUtil

This sample also demonstrates how two applets can have different contexts but both access the same string constant from a library.

The sample is composed of two applets (StringHandlingApp and StringUtilApp) and two libraries (StringHandlingLib and StringHandlingLibLocal). The libraries use string annotations to define string constants. The two applets use annotations to define their own set of string constants and to import string constants from the libraries.

In this sample, an applet such as StringHandlingApp uses string constants that it imports from one library, StringHandlingLibLocal, that are in turn imports of constants from another library, StringHandlingLib.

Follow one of these sets of instructions to run this sample:

- Running the StringHandlingApp Sample from Eclipse
- Running the StringHandlingApp Sample from the Command Line

Description of StringHandlingApp Applet

The StringHandlingApp applet uses javacardx.annotations.StringDef and javacardx.annotations.StringPool annotations. It defines its own set of string constants and also imports a string constant from the main library, StringHandlingLib. The StringHandlingApp applet demonstrates how you can use two applets in different contexts both importing a single string constant from a common library. The StringHandlingApp applet imports and uses one of the same string constants from StringHandlingLib as the StringUtilApp applet.

The StringHandlingApp applet imports a string constant from the StringHandlingLib, and also defines string constants for itself. When the StringHandlingApp is selected, the process method uses the test methods defined in the StringHandlingLib library. If the results from each of the tested methods match the expected string constants defined in the applet, it creates a response message containing a **Hello World!** message with a copy of the incoming message appended to the end. In the case that the tested methods do not produce the expected outcome, it sends a message containing the header bytes from the buffer with a copy of the incoming message appended to the end.

Description of StringUtilApp Applet

StringUtilApp uses javacardx.framework.string.StringUtil class and combines the use of:

- String annotations
- String constants from the StringHandlingLib and StringHandlingLibLocal libraries

- Methods from the StringUtil class

It imports string constants from one library, StringHandlingLibLocal, that are in turn imports of constants from another library, StringHandlingLib. In addition, it also helps demonstrate how two applets can have different contexts but both access the same string constant from a common library. It imports and uses one of the same string constants from StringHandlingLib as the StringHandlingApp applet.

The StringUtilApp process method handles APDUs containing a command string composed of a command type and optional arguments. It sends a response APDU based on the command string it received. Contained in this applet's string pool are string constants defining stored items. Command and response strings are represented as a series of bytes following a utf-8 representation of strings.

Command String Requirements

The following are requirements for a valid command string:

- Command strings must be terminated by a period.
- Command type names must be separated from their arguments by a space.
- Command type names (welcome, contacts, and settings) are case insensitive.
- Arguments for Contacts or Settings command types are 1 or 2. See Table 4-1.
- Optional second argument of 1 or 2 can be used for the Settings command type. See Table 4-1.

Table 4-1 Valid Command String Type and Argument Combinations

Command Types	Command Arguments	Optional Arguments
Welcome	none	none
Contacts	1 or 2	none
Settings	1 or 2	1 or 2

To demonstrate the applet's command string functionality, the following examples are provided as string versions of the byte sequences of valid command strings:

> **Note:**
>
> Valid command type names used in command strings are case insensitive.

- `Welcome.`
- `Settings 1.`
- `contacts 2.`
- `Settings 2 2.`

Response String Description

Response strings are automatically formatted by the applet when a command APDU is received. You must follow the requirements for creating a valid command string (see

Command String Requirements) to send to the applet or the applet does not produce the desired results. While you do not create the response strings, the following describes the responses you can expect from the applet:

- If the command string is invalid, then a default response string is sent.
- If the command string contains the `Welcome` command type, then a welcome response string is sent.
- If the command string contains `Contacts` or `Settings` command types with arguments, then a string is sent that corresponds to the arguments and command type received. This response string is composed of a comma separated name and value pair.

To demonstrate the applet's response string functionality, the following examples are response string versions of byte sequences that correspond to the example command strings in Command String Requirements:

- `Hello California!`
- `AutoCorrect, Off`
- `John Adams, John.Adams@123.com`
- `Wifi, On`

Examples of Process Method Handling of APDUs Containing a Command String

The following are examples of process methods `StringUtilApp` applet uses when processing the command strings containing default values and settings for `Contacts` and `Settings`:

- `Contacts` with a name and an e-mail address

  When `StringUtilApp` receives a command for `Contacts`, it sends a response message with the contact name corresponding to the number in the command argument along with the associated e-mail address value for that name.
- `Settings` with arguments

  When a `Settings` command is received with only one argument, the setting corresponding to the number in the command argument and its default value are sent in the response message.

  If the `Settings` command contains two arguments, the second argument signifies the state in which that setting should be placed. The applet then responds with the name of the setting and the value that was selected in the command.

Description of StringHandlingLib and StringHandlingLibLocal Libraries

The following libraries, `StringHandlingLib` and `StringHandlingLibLocal`, use string annotations to define string constants:

- `StringHandlingLib` - The main library is used by both applets and contains the following:
  - String constants for a default location, a hello greeting, and an error message
  - Examples of `substring`, `startsWith`, and `endsWith` methods implemented by using the `offsetByCodePoints` method from `StringUtil`

  - Test methods for the `substring`, `startsWith`, `endsWith`, and `offsetByCodePoints` methods
- `StringHandlingLibLocal` - The local library is an example of a library that contains location specific string constants.

  As used in `StringUtilApp`, the local library provides an example of how a library:

  - Controls the formatting of input and output strings including delimiters for arguments and command string terminators
  - Provides the location used in the welcome message and a default error message

> **Note:**
>
> The default error message demonstrates how a library can use string constants from another library and how an applet can use string constants from either the main or the local library.

Running the StringHandlingApp Sample from Eclipse

The StringHandlingApp sample consists of three Java Card projects: `StringHandlingApp`, `StringHandlingAppLib`, and `StringHandlingAppLibLocal`.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the following Java Card projects into your workspace: `StringHandlingApp`, `StringHandlingAppLib`, and `StringHandlingAppLibLocal`. If the builds don't start automatically, start them manually.

   About the build order: the `StringHandlingApp` project depends on both `StringHandlingAppLib` and `StringHandlingAppLibLocal`. `StringHandlingAppLibLocal` depends on `StringHandlingAppLib`. These dependencies define the order in which the projects are built by Eclipse: first `StringHandlingAppLib`, then `StringHandlingAppLibLocal`, and finally `StringHandlingApp`. In the `StringHandlingApp` project, the packages are built in order of their AID values: first `com.sun.jcclassic.samples.stringapp`, then `com.sun.jcclassic.samples.stringutilapp`.

2. Perform the following steps for the `StringHandlingApp` project:

   a. In the **Package Explorer** view, click the **StringHandlingApp** project.

   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.

   c. Select a CAP file from the list that appears in the **Java Card CAP Files** page.

   d. Click **StringHandlingApp1** and select **Edit**.

   e. In the **Edit** mode, select **Compact CAP File**.

   f. Click **Next>**.

   g. Select **ScriptGen** slide and select the **Supress "PowerUp; APDU command at the beginning of CAP script** and **Supress "PowerDown; APDU command at the end of CAP script** check boxes.
   h. Click **Finish**.
   i. Repeat Steps a to c.
   j. Click **StringHandlingApp2** and select **Edit**.
   k. Repeat Steps e to g.
   l. Click **Finish**, and select **Apply** and **Close**.
3. Perform the following steps for the `StringHandlingAppLib` project:
   a. In the **Package Explorer** view, click the **StringHandlingAppLib** project.
   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the **Java Card CAP Files** page.
   d. Click **StringHandlingAppLib** and select **Edit**.
   e. In the **Edit** mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Supress "PowerDown; APDU command at the end of CAP script** check box.
   h. Click **Finish**, and select **Apply** and **Close**.
4. Perform the following steps for the `StringHandlingAppLibLocal` project:
   a. In the **Package Explorer** view, click the **StringHandlingAppLibLocal** project.
   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the **Java Card CAP Files** page.
   d. Click **StringHandlingAppLibLocal** and select **Edit**.
   e. In the **Edit** mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Supress "PowerUp; APDU command at the beginning of CAP script** and **Supress "PowerDown; APDU command at the end of CAP script** check boxes.
   h. Click **Finish**, and select **Apply** and **Close**.
5. Rebuild the modified projects: select **Project** and **Clean**, and from the Clean dialog, select the three projects that you just modified.

   Eclipse builds the projects again.
6. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:
   a. Clear the **Input file with EEPROM data**, the **Output file for EEPROM data**, and the **Combined (input and output) file for EEPROM data** fields.
   b. Clear **Do not open APDU console**.
   c. Click **OK**.

7. In Java Card View, right-click on Sample_Device and select **Start**.

   The simulator starts and you can see that Sample_Device console is created.

8. Execute the scripts in the following order:

   - `cap-StringHandlingApp`
   - `cap-StringHandlingAppLibLocal`
   - `cap-StringHandlingApp1`
   - `cap-StringHandlingApp2`
   - `stringhandlingapp`

Now you can compare the console output with the contents of `test.expected.output` file.

Running the StringHandlingApp Sample from the Command Line

To run the StringHandlingApp sample:

1. Open a Command Prompt window, navigate to the `JC_HOME_SIMULATOR\bin` directory, and start the simulator by entering the following command at the command prompt:

   ```
   cref -o stringapp
   ```

2. Open a second Command Prompt window, set `ANT_HOME`, `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` as environment variables, navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\StringHandlingApp\lib` directory, and at the command prompt, enter:

   ```
   ant all
   ```

3. In the first Command Prompt window, restart the simulator by typing:

   ```
   cref -i stringapp -o stringapp
   ```

4. In the second Command Prompt window, navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\StringHandlingApp\liblocal` directory, and at the command prompt, enter:

   ```
   ant all
   ```

5. In the first Command Prompt window, start the simulator by entering the following command at the command prompt:

   ```
   cref -i stringapp
   ```

6. In the second Command Prompt window, navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\StringHandlingApp\applet` directory and, at the command prompt, enter:

   ```
   ant all
   ```

7. Verify that the contents of the output file, `default.output`, in the `applet` directory are the same as the contents of the `test.expected.out` file.

SecureRMIPurse Sample

This sample is only included in bundles with cryptography extensions.

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, except that it creates and registers an instance of MySecurityService.

The client-side program, SecurePurseClient, is similar to the non-secure case, except 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 command described in Table 4-2.

Table 4-2 Authenticate User Command

CLA_AUTH	INS_AUTH	P1 field	P2 field	LC field	PIN (byte 1)	PIN (byte 2)
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.

Follow one of these sets of instructions to run this sample:

- Running the SecureRMIPurse Sample in Eclipse
- Running the SecureRMIPurse Sample from the Command Line

Running the SecureRMIPurse Sample in Eclipse

The `SecureRMIPurse` sample is the same as `RMIPurse` , but with an added security service. It consists of two projects: a Java Card project with the Java Card applet and a Java SE project with the Java application that is designed to communicate with the applet.

1. If you haven’t already, create the launch configuration for the rmic tool using the instructions here: Running the RMIPurse Sample in Eclipse.
2. Follow the rest of the instructions in Running the RMIPurse Sample in Eclipse, but substitute `SecureRMIPurse` wherever you see `RMIPurse`.

When you have completed all the steps, and you see the application output in the console, compare it with the contents of `securermidemo.expected.out`.

Running the SecureRMIPurse Sample from the Command Line

To run SecureRMIPurse:

1. Open a Command Prompt window and perform the following:
   1. Navigate to the `JC_HOME_SIMULATOR\bin` directory.
   2. Start the simulator by using the following command at the command prompt:

      ```
      cref -o demoee
      ```

   Starting the simulator with the `-o` option causes the simulator to save the EEPROM contents to a file named `demoee`.

2. Open a second Command Prompt window and perform the following:
   1. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.
   2. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\SecureRMIPurse\applet` directory.
   3. 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 simulator by using the following command:

   ```
   cref -i demoee
   ```

4. In the applet Command Prompt window, perform the following:
    a. Navigate to the `JC_HOME_SIMULATOR\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.actual.output` file in the `client` directory are the same as the contents of the `securermidemo.expected.out` file in the `SecureRMIPurse` directory.

SignatureMessageRecovery Sample

> **Note:**
>
> This sample is only included in bundles with cryptography extensions.

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.

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 sequence of events resulting from running the first script, `sigMsgFullRec.scr`, 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.

The second script, `sigMsgPartRec.scr`, demonstrates a 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 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.

Message Recovery Order of Operations for Signing

The order of operations for signing is as follows:

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.

Message Recovery Order of Operations for Verifying

The order of operations for verifying is as follows

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.

Running the SignatureMessageRecovery Sample in Eclipse

In this sample we create two run configurations for the same project, to run a different pair of scripts.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `SignatureMessageRecovery` project into your workspace. If the build doesn't start automatically, start it manually.
2. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:
   a. Select **Do not open APDU console**.
   b. Click **OK**.
3. Before you configure, run, and start any script, you must change the `PowerDown` parameters for generating the script files. Otherwise, the simulator goes into the

PowerDown mode after running the `cap-SignatureMessageRecovery.script`, and interrupts the execution of the following script files. To change the `PowerDown` parameters:

   a. In the **Package Explorer** view, click `SignatureMessageRecovery` project.
   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the Java Card CAP Files page.
   d. Click **SignatureMessageRecovery** and select **Edit**.
   e. In the Edit mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Suppress "PowerDown;" APDU command at the end of CAP script** check box.
   h. Click **Finish** and select **Apply** and **Close**.

4. Create the first Run Configuration for this project. In the top menu, select **Run** and **Run Configurations...**
5. In the Run Configurations dialog:
   a. Right-click on **Java Card Project Run** and select **New**.
   b. In the **Name** field, enter `SignatureMessageRecovery_PartRec`
   c. Click **Browse...**, select the `SignatureMessageRecovery` project, and click **OK**.
   d. Select **Start simulator**.
   e. In the **Scripts to be executed on simulator** list box, add the following scripts:
      - `cap-com.sun.jcclassic.samples.signaturemessagerecovery.script` from `JC_HOME_SIMULATOR\samples\classic_applets\SignatureMessageRecovery\applet\apdu-scripts`
      - `sigMsgPartRec.scr` from the same directory.
   f. Click **Apply** and **Close**
6. Create the second Run Configuration for this project. In the top menu, select **Run** and **Run Configurations...**
7. In the Run Configurations dialog:
   a. Right-click on **Java Card Project Run** and select **New**.
   b. In the **Name** field, enter `SignatureMessageRecovery_FullRec`
   c. Click **Browse...**, select the `SignatureMessageRecovery` project, and click **OK**.
   d. Select **Start simulator**.
   e. In the **Scripts to be executed on simulator** list box, add the following scripts:
      - `cap-com.sun.jcclassic.samples.signaturemessagerecovery.script` from

`JC_HOME_SIMULATOR\samples\classic_applets\SignatureMessageRecovery\applet\apdu-scripts`

- `sigMsgFullRec.scr` from the same directory. (Note that this script is different from the first Run Configuration that you created)

f. Click **Apply** and **Close**

8. In the top menu, select **Run** and **Run Configurations...**, select **SignatureMessageRecovery_PartRec**, click **Run**.

   Compare the output with the contents of the `sigMsgPartRec.expected.output` file.

9. In the top menu, select **Run** and **Run Configurations...**, select **SignatureMessageRecovery_FullRec**, click **Run**.

   Compare the output with the contents of the `sigMsgFullRec.expected.output` file.

Running the SignatureMessageRecovery Sample from the Command Line

To run the SignatureMessageRecovery sample:

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

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

   b. Start the simulator 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. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.

   b. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\SignatureMessageRecovery\applet` directory.

   c. 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 simulator 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.

ArrayViews Sample

The ArrayViews sample demonstrates a client application and a server application sharing data using array views.

To run the sample, see Running the ArrayViews Sample from the Command Line.

Running the ArrayViews Sample from the Command Line

Perform the following steps to run the ArrayViews sample from the command line.

To run the ArrayViews sample:

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

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

   b. Start the simulator 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. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.

   b. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\ArrayViews\applet` directory.

   c. 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 Arrayviews.expected.output file.

CertHandling Sample

The CertHandling Sample demonstrates the use of static resources and certificate API to parse and verify a certificate.

To run the sample, see Running the CertHandling Sample from the Command Line.

Running the CertHandling Sample from the Command Line

Perform the following steps to run the CertHandling sample from the command line.

To run the CertHandling sample:

1. Open a Command Prompt window and perform the following:
   a. Navigate to the JC_HOME_SIMULATOR\bin directory.
   b. Start the simulator 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. Set ANT_HOME (path to ant install folder), JC_HOME_TOOLS and JC_HOME_SIMULATOR (path to JCDK install folder) as environment variables.
   b. Navigate to the JC_HOME_SIMULATOR\samples\classic_applets\CertHandling\applet directory.
   c. 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 CertHandling.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.

- `JavaPurseCrypto` Sample - Demonstrates the use of a DES MAC algorithm.

  This sample is only included in bundles with cryptography extensions. See JavaPurseCrypto Sample.

- `PurseWithLoyalty` Sample Application - Demonstrate the use of shareable interfaces.

  See JavaPurse Sample Application.

- `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 with cryptography extensions. See Transit Sample.

Biometry Sample Application

In this sample, a user password is enrolled on the card and a candidate password is matched against the enrolled password. The sample demonstrates the basic functionality of the biometric API. In the sample, everything works well. Non-sample code should be prepared to handle errors that may occur during the enrollment process or the matching process, including a card-blocked state, or a non-initialized state. See How the Biometric API Works.

1. The off-card tool takes a hard coded password and sends it to the card for enrollment. For this sample, the off-card tool is a simple apdutool script used for both enrolling and matching.

   The applet selected on-card is the `SampleBioServer` applet. See SampleBioServer Class.

2. The `SampleBioServer` applet stores the password as the reference template with five tries allowed before block.

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.

4. The script sends the same password that was 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.

Follow one of these sets of instructions to run this sample:

- Running the Biometry Sample in Eclipse
- Run the Biometry Sample from the Command Line

SampleBioServer Class

This class represents the `BioServer` applet on the card. This class is the interface to the on-card and off-card clients for the biometric functionality on the card.

It communicates with off-card clients with APDUs, and with on-card client applets with an implementation of `SharedBioTemplate`. This class causes the enrolling of the biometric password while communicating with an off-card tool that sends the password to the BioServer.

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 provides 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

Biometric verification must happen on-card for security reasons. The card cannot send out a person's biometric information or a PIN for verification to be done off-card; 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). 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 Oracle 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*

Running the Biometry Sample in Eclipse

The Biometry sample consists of two Java Card projects: Biometry_Client and Biometry_Server. We will run them without the APDU console.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `Biometry_Client` and `Biometry_Server` projects into your workspace. If the builds don't start automatically, start them manually.
2. Before you start any script, you must change the `PowerDown` and `PowerUp` parameters for generating the script files. Otherwise, the simulator goes into the PowerDown mode or interrupt the execution by throwing error messages.

   To change the `PowerDown` parameters in the `Biometry_Server` project :

   a. In the **Package Explorer** view, click `Biometry_Server` Java project.
   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the Java Card CAP Files page.
   d. Click **Biometry_Server** and select **Edit**.
   e. In the Edit mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Suppress "PowerDown;" APDU command at the end of CAP script** check box.
   h. Click **Finish** and select **Apply** and **Close**.

   To change the `PowerDown` parameters in the `Biometry_Client` project :

   a. In the **Package Explorer** view, click `Biometry_Client` Java project.
   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the Java Card CAP Files page.
   d. Click **Biometry_Client** and select **Edit**.
   e. In the Edit mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Supress "PowerUp; APDU command at the beginning of CAP script** and **Suppress "PowerDown;" APDU command at the end of CAP script** check boxes.
   h. Click **Finish** and select **Apply** and **Close**.
3. Right-click on `Biometry_Client`, select **Properties**, select **Run/Debug Settings**, click **New...**. The Select Configuration Type window opens. Select **Java Card Project Run**, click **OK**.
4. In the Edit Configuration dialog:

   a. In the **Name** field, enter `Biometry`
   b. Select **Start simulator**.
   c. In the **Scripts to be executed on simulator** list box, add the following scripts:

    - cap-com.sun.jcclassic.samples.biometryserver.script from Biometry_Server
    - cap-com.sun.jcclassic.samples.biometryclient.script from Biometry_Client
    - biometryEnroll.scr from Biometry_Client
    - biometryMatch.scr from Biometry_Client

   d. Click **Apply** and **Close**

5. In Java Card View, double-click on Sample_Device. In the Properties for Sample_Device dialog, select the **CREF** tab:

   a. Select **Do not open APDU console**.

   b. Click **OK**.

6. In the top menu, select **Run** and **Run Configurations...**, select **Biometry** and click **Run**.

   Compare the output with the contents of the biometry-client.expected.out file.

Running the Biometry Sample from the Command Line

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

   a. Navigate to the JC_HOME_SIMULATOR\bin directory.

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

      ```
      cref -o e2p
      ```

   The simulator saves the EEPROM contents to a file named e2p.

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

   a. Set ANT_HOME (path to ant install folder), JC_HOME_TOOLS and JC_HOME_SIMULATOR (path to JCDK install folder) as environment variables.

   b. Navigate to the JC_HOME_SIMULATOR\samples\reference_apps\Biometry\Server\applet directory.

   c. 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 simulator by using the following command:

   ```
   cref -i e2p
   ```

   The simulator uses the contents of the e2p file to initialize the EEPROM.

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

   a. Navigate to the JC_HOME_SIMULATOR\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.

Running the JavaPurse Sample in Eclipse

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `JavaPurse` project into your workspace.

   If the build doesn't start automatically, start it manually.

2. Before you start any script, you must change the `PowerDown` and `PowerUp` parameters for generating the script files. Otherwise, the simulator goes into the PowerDown mode or interrupt the execution by throwing error messages. To change the `PowerDown` parameters:

   a. In the **Package Explorer** view, click the `JavaPurse`Java project.

   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the Java Card CAP Files page.
   d. Click **SampleLibrary** and select **Edit**.
   e. In the Edit mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Suppress "PowerDown;" APDU command at the end of CAP script** check box.
   h. Click **Finish**.
   i. Click **JavaLoyality** and select **Edit**.
   j. In the Edit mode, select **Compact CAP File**.
   k. Click **Next>**.
   l. Select **ScriptGen** slide and select the **Supress "PowerUp; APDU command at the beginning of CAP script** and **Supress "PowerDown; APDU command at the end of CAP script** check boxes.
   m. Click **Finish**.
   n. Click **JavaPurse** and select **Edit**.
   o. In the Edit mode, select **Compact CAP File**.
   p. Click **Next>**.
   q. Select **ScriptGen** slide and select the **Supress "PowerUp; APDU command at the beginning of CAP script** and **Supress "PowerDown; APDU command at the end of CAP script** check boxes.
   r. Click **Finish** and select **Apply** and **Close**.

3. In the **Package Explorer** view, click the `JavaPurse` project and then right-click on the Java Card project and build the project manually.
4. Select the `JavaPurse` project, press **Alt**+**Enter**, select **Run/Debug Settings**, and click **New**. In the Select Configuration Type window select **Java Card Project Run** and click **OK**.
5. In the Edit Configuration window do the following:
   a. Enter `JavaPurse` in the **Name** field
   b. Select the **Start simulator** check box
   c. Add the following scripts to the listbox:
      - `cap-SampleLibrary.script`
      - `cap-JavaLoyalty.script`
      - `cap-JavaPurse.script`
      - `jp.scr`
   d. Click **Apply** and **OK** to close the Edit Configuration window.
   e. Click **OK** to close the Properties window.
6. In Java Card View, double-click **Sample_Device**. In the Properties for Sample_Device dialog, select the **CREF** tab:

   a. Select **Do not open APDU console**.

   b. Click **OK**.

7. Select **Run** and **Run Configurations**. Select `JavaPurse` then click **Run**.

   Compare the console output with the content of `javapurse.expected.output` file.

Running the JavaPurse Sample from the Command Line

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

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

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

   ```
   cref
   ```

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

   a. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.

   b. Navigate to the `JC_HOME_SIMULATOR\samples\reference_apps\PurseWithLoyalty\JavaPurse\applet` directory.

   c. 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.

JavaPurseExtCap Sample

The `JavaPurseExtCap` Sample starts from the existing `JavaPurse` sample (which consists of two applets and a library) and demonstrates the process to bundle them together and deploy in a single extended CAP file.

To run the sample, see Running the JavaPurseExtCap Sample from the Command Line.

Running the JavaPurseExtCap Sample from the Command Line

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

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

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

```
cref
```

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

   a. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.

   b. Navigate to the `JC_HOME_SIMULATOR\samples\reference_apps\PurseWithLoyalty\JavaPurseExtCap\applet` directory.

   c. 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

> **Note:**
>
> This sample is only included in bundles with cryptography extensions.

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 for additional information about the `JavaLoyalty` applet.

Running the JavaPurseCrypto Sample in Eclipse

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `JavaPurseCrypto` project into your workspace.

   If the build doesn't start automatically, start it manually.
2. Select the `JavaPurseCrypto` project, press **Alt**+**Enter**, select **Run/Debug Settings**, and click **New**. In the Select Configuration Type window select **Java Card Project Run** and click **OK**.
3. Before you start any script, you must change the `PowerDown` and `PowerUp` parameters for generating the script files. Otherwise, the simulator goes into the PowerDown mode or interrupt the execution by throwing error messages. To change the `PowerDown` parameters:
   a. In the **Package Explorer** view, click `JavaPurseCrypto`Java project.
   b. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
   c. Select a CAP file from the list that appears in the Java Card CAP Files page.
   d. Click **SampleLibrary** and select **Edit**.
   e. In the Edit mode, select **Compact CAP File**.
   f. Click **Next>**.
   g. Select **ScriptGen** slide and select the **Suppress "PowerDown;" APDU command at the end of CAP script** check box.
   h. Click **Finish**.
   i. Click **JavaLoyality** and select **Edit**.
   j. In the Edit mode, select **Compact CAP File**.
   k. Click **Next>**.
   l. Select **ScriptGen** slide and select the **Supress "PowerUp; APDU command at the beginning of CAP script** and **Supress "PowerDown; APDU command at the end of CAP script** check boxes.
   m. Click **Finish**.
   n. Click **JavaPurse** and select **Edit**.
   o. In the Edit mode, select **Compact CAP File**.
   p. Click **Next>**.
   q. Select **ScriptGen** slide and select the **Supress "PowerUp; APDU command at the beginning of CAP script** and **Supress "PowerDown; APDU command at the end of CAP script** check boxes.
   r. Click **Finish** and select **Apply** and **Close**.
4. In the Edit Configuration window do the following:
   a. Enter `JavaPurseCrypto` in the **Name** field
   b. Select the **Start simulator** checkbox

c. Add the following scripts to the listbox:
   - `cap-SampleLibrary.script`
   - `cap-JavaLoyalty.script`
   - `cap-JavaPurse.script`
   - `jpcrypto.scr`

d. Click **Apply** and **OK** to close the Edit Configuration window.

e. Click **OK** to close the Properties window.

5. In the Java Card View, double-click **Sample_Device**. In the Properties for Sample_Device dialog, select the **CREF** tab:

   a. Select **Do not open APDU console**.

   b. Click **OK**.

6. Select **Run** and **Run Configurations**. Select `JavaPurse` then click **Run**.

   Compare the console output with the content of `javapursecrypto.expected.output` file.

Running the JavaPurseCrypto Sample from the Command Line

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

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

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

   ```
   cref
   ```

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

   a. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.

   b. Navigate to the `JC_HOME_SIMULATOR\samples\reference_apps\PurseWithLoyalty\JavaPurseCrypto\applet` directory.

   c. 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

> **Note:**
>
> This sample is only included in bundles with cryptography extensions.

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 varies from that of the expected output files for the following instructions:

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

Running the Transit Sample in Eclipse

The Transit sample consists of two projects: a Java Card project with the Java Card applet and a Java SE project with the Java application that is designed to communicate with the applet.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Import the `Transit_Applet` Java Card project and `Transit_Client` Java project into your workspace.

   You can import both projects in the same Import wizard. If the builds don't start automatically, start them manually.

   The `Transit_Applet` project build creates `apdu_scripts` and `deliverables` directories.
2. In Java Card View, double-click **Sample_Device**. In the Properties for Sample_Device dialog, select the **CREF** tab:
   a. In the **Combined (input and output) file for EEPROM data** field, type a file name to be used for saving EEPROM between simulator sessions, such as `TransitCard.eeprom`. The file will be automatically created in the `bin` directory. Later, after the sample run, you can safely delete it.
   b. Clear the **Input file with EEPROM data**, and the **Output File for EEPROM data fields**.
   c. Select **Do not open APDU console**.
   d. Click **OK**.
3. In Java Card View, right-click `Sample_Device` and select **Start**.

The simulator starts and you can see that the Sample_Device console is created.

4. In the Sample_Device console toolbar, click the **Select Script** drop-down button and select `cap-com.sun.jcclassic.samples.transit`.

   Wait until the script execution completes and CMD> prompt is displayed.

5. Click the **Select Script** drop-down button again and select the `TransitDemoFooter_notransitkey` script.

   The script executes and the simulator stops. Verify that the last script finished successfully (the last APDU command got response with SW1: 90, SW2: 00).

6. Create run configurations: `run1`
   a. Right-click the `Transit_Client` project, select **Properties**, select **Run/Debug Settings**, and click **New**.
   b. In the Select Configuration Type window, select **Java Application**, click **OK**.
   c. Set the **Name** field to `run1`.
   d. Set the **Main class** to `com.sun.jcclassic.clients.transit.POSTerminal`
   e. Select the **Arguments** tab
   f. Set the **Program arguments** to

      ```
      -k FFFFFFFFFFFFFFFF -- VERIFY 12345 CREDIT 99 GET_BALANCE
      ```
   g. Click **Apply** and **OK** to close the window.

7. Create run configurations: `run2`
   a. In the Select Configuration Type window, select **Java Application**, click **OK**.
   b. Set the **Name** field to `run2`.
   c. Set the **Main class** to `com.com.sun.jcclassic.clients.transit.TransitTerminal`
   d. Select the **Arguments** tab
   e. Set the **Program arguments** to

      ```
      -k FFFFFFFFFFFFFFFF -- PROCESS_ENTRY 999
      ```
   f. Click **Apply** and **OK** to close the window.

8. Create run configurations: `run3`
   a. In the Select Configuration Type window, select **Java Application**, click **OK**.
   b. Set the **Name** field to `run3`.
   c. Set the **Main class** to `com.sun.jcclassic.clients.transit.TransitTerminal`
   d. Select the **Arguments** tab
   e. Set the **Program arguments** to

      ```
      -k FFFFFFFFFFFFFFFF -- PROCESS_EXIT 10
      ```
   f. Click **Apply** and **OK** to close the window.

9. Create run configurations: `run4`
   a. In the Select Configuration Type window, select **Java Application**, click **OK**.

   b. Set the **Name** field to `run4`.
   c. Set the **Main class** to `com.sun.jcclassic.clients.transit.POSTerminal`
   d. Select the **Arguments** tab
   e. Set the **Program arguments** to

      `-k FFFFFFFFFFFFFFFF -- VERIFY 12345 GET_BALANCE`

   f. Click **Apply** and **OK** to close the window.

10. In the Java Card View, double-click **Sample_Device**. In the Properties for Sample_Device dialog, select the **CREF** tab:
    a. Select **Do not open APDU console**.
    b. Click **OK**.
11. In Java Card View, right-click **Sample_Device** and select **Start**.

    The simulator starts and you can see the output in the Console view.

12. In the Eclipse top menu, select **Run** and **Run Configurations**. Expand the `Java Application` entry if necessary, select **run1**, and then click **Run**.
13. Verify that the contents of the console 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 console varies from the `TransitClient_1.expected.output` file for the following instructions:

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

14. In Java Card View, right-click **Sample_Device** and select **Start**.

    The simulator starts and you can see the output in the Console view.

15. In the Eclipse top menu, select **Run** and **Run Configurations**. Select **run2**, and then click **Run**.
16. Verify that the contents of the console 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 console varies from the `TransitClient_2.expected.output` file for the following instructions:

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

17. In Java Card View, right-click **Sample_Device** and select **Start**.

    The simulator starts and you can see the output in the Console view.

18. In the Eclipse top menu, select **Run** and **Run Configurations**. Select **run3**, and then click **Run**.
19. Verify that the contents of the console 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 console varies from the `TransitClient_3.expected.output` file for the following instructions:

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

20. In Java Card View, right-click **Sample_Device** and select **Start**.

    The simulator starts and you can see the output in the Console view.

21. In the Eclipse top menu, select **Run** and **Run Configurations**. Select **run4**, and then click **Run**.

22. Verify that the contents of the console 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 console varies from the `TransitClient_4.expected.output` file for the following instructions:

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

Running the Transit Sample from the Command Line

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_HOME_SIMULATOR\bin` directory.

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

   ```
   cref -o transitCard
   ```

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

   a. Set `ANT_HOME` (path to ant install folder), `JC_HOME_TOOLS` and `JC_HOME_SIMULATOR` (path to JCDK install folder) as environment variables.

   b. Navigate to the `JC_HOME_SIMULATOR\samples\reference_apps\Transit\Transit\applet` directory.

   c. 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 simulator by using the following command:

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

   Starting the simulator with the `-i transitCard -o transitCard` options and filenames causes the simulator to use the contents of the `transitCard` file to initialize the EEPROM and to save the EEPROM contents to a file named `transitCard`.

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

   a. Navigate to the `JC_HOME_SIMULATOR\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 varies 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 simulator 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 varies 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 simulator 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 varies 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 simulator 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 varies from the `TransitClient_4.expected.output` file for the following instructions:

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

5 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. This chapter contains the following sections:

Overview of Converting and Exporting Java Class Files

The Converter processes all of the Java class files that make up an application (or a library) and creates a binary file (CAP file) that can be deployed and loaded on a Java Card platform. It also produces other files (export files and JCA files) that are used in the development and deployment process. The CAP file contains a manifest file that provides human-readable information about its content. See Working with CAP Files and Using Export Files, for more information.

The Java Card Platform Specification, Version 3.1 defines a new version of the CAP file that supports the following formats:

- Compact CAP file format - A compact CAP file contains a single Java Package, a method component of maximum 64 K, and may contain static resources. It can represent an application or a shared library.
- Extended CAP file format - The extended CAP file format can contain multiple Java packages and a method component larger than 64 K. It gives control over which package should be exported as a shared library.

The compact CAP file is supported in all Java Card products and offers the binary backward compatibility with all previous formats. The extended CAP file is optionally supported in Java Card products, version 3.1 and above.

- See the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1* for more information on the CAP file and its format.
- See Chapter Programming for the Large Address Space for more information on using the Compact CAP file format to take advantage of large memory storage in secure elements.
- See Chapter Programming Large Java Card Applications With Multiple Packages for more information on using the Extended CAP file format.

Using the Converter in the Compact or Extended Format

The selection of the format (either compact or extended) depends on the following factors:

- **Application Design**
  - The number of Java packages included in the application - If the application (or library) only includes a single package, it can be converted into a compact file format.
  - The total code size - If the application code creates a method component with a size larger than 64 K, then the extended format is required.
- **Deployment Constraints**
  - Some libraries must remain private - If an application relies on libraries that must not be shared, the extended format can be used to support application made of multiple packages instead of refactoring the code and copying the library classes into the application package.
  - A shared library includes both a public API made of one or more exported packages and private implementation packages - This could be achieved using an extended format CAP file that contains both parts, keeping the implementation packages private, and deploying all packages in one CAP file.

The format (compact or extended) can be set using the Converter command line parameters. See Running the Converter, for more details about the command line parameters. See Programming for Multipackage Large CAP Files, for more details about the Extended CAP file.

Using the Converter for a Target Java Card Version

The Converter can be used to create CAP files for Java Card versions 3.0.4, 3.0.5 and 3.1.0, using the `–target [version]` command line parameter. If the CAP file needs to be deployed on multiple Java Card versions, use the oldest version (that is, the smallest version number) as the target.

Table 5-1 Converter Usage

API Version	Converter Usage	When to Use?	CAP File Format Generated
3.0.4	`-target 3.0.4`	Use this mode when the target platform is 3.0.4.	2.2 (compact)
3.0.5	`-target 3.0.5`	Use this mode when the target platform is 3.0.5.	2.2 (compact)
3.1.0	`-target 3.1.0 (default)`	Use this format when the target platform is 3.1.0 and the code size is less than 64 K, in one package.	2.3 (compact)
3.1.0	`-target 3.1.0 -config <file.json>`	Use this format when the target platform is 3.1.0 and the code is made of multiple packages or code size is greater than 64 K.	2.3 (extended)

Using the Converter to Generate a Mask

You might choose to convert packages that import other packages. If you are creating Java Card Assembly files to generate a mask file, then the major and minor version numbers of the imported packages must agree with the version number of the package that imports them.

> **Note:**
>
> Generating mask file is possible in source bundle only.

See Java Card Assembly Syntax Example, for more information on the Java Card Assembly file.

Setting Java Compiler Options

To set Java compiler options:

- Compile your class files with the Java Development Kit 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 than in other environments.
- The `LocalVariableTable` attribute is not generated.

Running the Converter

To run the Converter:

1. For the compact mode, enter either of the following commands at the command line to invoke the Converter:

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

   Or

   `converter.bat -config` <*filename*>

2. For the extended mode, enter the following command at the command line to invoke the Converter:

`converter.bat -config` *<filename.json>*

3. For showing the usage, enter either of the following commands at the command line to invoke the Converter:

   ```
   converter.bat -help
   ```

   Or

   ```
   converter.bat -help JSON
   ```

> **Note:**
>
> The `converter.bat` file used to invoke the Converter is a batch file that you must run from a working directory of *JC_HOME_TOOLS*`\bin` in order for the code to execute properly.

The Converter command line options described in Table 5-2 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 pre-defined export files of the packages being converted. The Converter looks for the export files 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 outputs one or more of the following files:
  - CAP file
  - JCA file
  - EXP export file
- Identify that a 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.).
- Specify a list of file paths from where the static resources are loaded by the Converter, if any.
- Specify the target Java Card platform version on which the CAP file generated should be loaded, if it is not the newest released version of the Java Card platform.

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

1. **Loads the packages -** If the `exportmap` option is set for any of the packages, the Converter loads that package from the export path (see Specifying an Export Map). It loads the class files of the Java packages and creates a data structures to represent these packages.

2. **Subset checking** - Checks for unsupported Java features in class files.
3. **Conversion** - Checks for consistency between the applet AIDs, package AIDs, CAP file AID (if present), and the imported package AIDs.
4. **Reference Checking** - Checks that all references are valid, internal referenced items are defined in the packages belonging to the CAP file, and import items are declared in the export files (see Using Export Files).

   The Converter creates the `JcImportTokenTable` to store tokens for import items (class, methods, and fields). If the Converter only generates export files, 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.
5. **Optimization** - Optimizes the byte code.
6. **Generates output** - Builds and outputs one EXP export file for each package and one JCA file for each package, checks for each package version in the export file against the version specified in the command line or in the config file. If the `-exportmap` option is used for a specific package in the command line or config file, the export file specified in the command line for that package must represent the same version as that of the package. The converter does not support upgrading the export file version.

   Before writing the export files 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 class root directory specified by the `-classdir` option. You can specify a different *root_dir* by using the `-d` option.

   The Converter generates only one CAP file. In the compact mode, the CAP file contains only one package and it is written to the path mentioned into the preceding example (*root_dir*\package_dir\\`javacard`). In the extended mode, the CAP file contains one or more packages and it is written into the following directory:

   output_dir\CAP_name\\`javacard`

   By default, the *output_dir* is the directory where the JSON configuration file, which used in the extended mode, is located. You can specify a different *output_dir* by defining a value for the `outputDir` field in the JSON configuration file.

Table 5-2   Converter Command Line Arguments

Option	Description
`-help`	Prints help message.
`-help JSON`	Prints a JSON definition file (schema), for the JSON configuration file to be used in extended mode. The JSON schema contains all of the fields that can be defined, the hierarchy of fields, field types, field descriptions, optionality, sample values, default values, and descriptions. The schema can be used (using various tools) for validating configuration files used for generating extended CAP files.
package-name	Fully-qualified name of the package to convert.
package-aid	5- to 16-decimal, hex or octal numbers separated by colons. Each of the numbers must be byte-length.

Table 5-2 (Cont.) Converter Command Line Arguments

Option	Description
major-version minor-version	User-defined version of the package.
`-applet` *AID class_name*	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.
`-classdir` *root-directory-of-class hierarchy*	Sets the root directory where the Converter looks for classes. If this option is not specified, the Converter uses the current user directory as the root.
`-d` *root-directory-for-output*	Sets the root directory for output.
`-debug`	Generates the optional debug component of a CAP file. If the `-mask` option is also specified, the file `debug.msk` is 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 looks for the export file in the `exportpath`.
`-exportpath` *list-of-directories*	Specifies the root directories in which the Converter looks 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`	Indicates that the converted code is intended to be used to create a binary 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. This option can be used in conjunction with `-out CAP`, only if `-debug` is selected, to typically generate a CAP with debug component and use it to debug platform classes. Such CAP is not intended to be loaded on a platform and will fail verification if it contains native methods.
`-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 .
`-nowarn`	Instructs the Converter not to report warning messages.
`-out` [*CAP*] [*EXP*] [*JCA*]	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` *value*	Keystore to use in signing

Table 5-2 (Cont.) Converter Command Line Arguments

Option	Description
`-storepass` *value*	Keystore password
`-alias` *value*	Keystore alias to use in signing
`-passkey` *value*	Alias password
`-useproxyclass`	Cannot be specified with `keepproxysource`. 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. 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`.cap` components. New descriptors are created. Dynamically-loaded-classes attributes need to be recomputed based on the new Proxy class file names.
`-usecapcomponents`	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: *application-classes-dir*`/`*application-classes*`/javacard/*.cap`
`-keepproxysource` *directory*	Cannot be used with `-useproxyclass`. Creates the proxy source files and other stub files in the specified *directory*. The converter also builds CAP files as usual in the specified output directory. Supports customizing the proxy files generated by the converter. Requests the converter retain the intermediate proxy class source code in the specified directory and the source code of the associated stub classes representing the dependent external classes using the hierarchical directory structure of the Java package name(s).
`-resourcepath` *<id1>*`:`*<resource_path1>*`,`*<id2>*`:`*<resource_path2>*`,...`	Specifies the list of static resources that can be loaded into the CAP file that is generated by the Converter (in the compact mode). The entries in the list are delimited by the `","` character. Each entry in the list contains two parameters delimited by the `":"` character. The first parameter is an integer representing the *id* of the static resource and the second parameter is the *path* to the file, which has the actual binary content for the static resource. The *path* must be a valid path to a file on the disk for which the Converter should have read access.

Table 5-2 (Cont.) Converter Command Line Arguments

Option	Description
`- target` *<platform version>*	Specifies the Java Card platform version on which the CAP file that is generated by the Converter (in the compact mode) is loaded. If the target is not specified in the converter, the default value would be the latest release version, that is, 3.1.0. Other valid values for the current release are, 3.0.4 and 3.0.5. If you are not using the target option or if you are using a target greater than 3.0.5, the 2.3 version CAP files are generated. Else, 2.2 or 2.1 version CAP files are generated, depending on the features (debugging or RMI). Also, for the current release, the *platform api_export_files* directory is not required in the `-exportpath` option. The directory for the platform API export files is chosen based on the `-target` option as follows: *JC_HOME_TOOLS*\api_export_files_<platform version>.

Using Delimiters with Command Line Options

To use delimiters with command line options:

- Add a double quote (") around command line option arguments that contain a space symbol.

In the following sample command line, the converter checks for export files in the`.\export files`, *JC_HOME_TOOLS*`\api_export_files_3.0.5`, and current directories.

```
converter -target 3.0.5 -exportpath ".\export files";.

MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

Using a Command Configuration File in Compact Mode

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.

To use a command configuration file:

1. Enter the command line arguments and options in a text-format configuration file.
2. Use double quote (") delimiters for the command line options that require arguments in 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 were placed in a configuration file, the result would look like this:

   ```
   converter -target 3.0.5 -exportpath ".\export files";.

   MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
   ```
3. Specify the configuration file in the command line when you run the Converter.

The syntax to specify a configuration file is:

`converter –config` *configurationfile name*

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

If the name of the configuration file has the `.json` extension, the extended mode is activated, else the compact mode is used.

Using a JSON Configuration File for Converter in the Extended Mode

In the extended mode, the Converter tool generates extended CAP files from one or multiple Java packages.

To run the Converter in the extended mode, use a JSON configuration file with the `-config` option. The JSON file includes fields and options that are used in the compact mode, however most of these fields and options are associated with each package contained in the CAP file.

The configuration in the JSON file is a JSON object with a single field, `inputConfig`. All other fields are defined inside this field at different levels of hierarchy. The description of levels follow:

- CAP file - Includes options for the entire CAP file.
- Package - Includes options specific to each package in the CAP file.
- Applet - Includes options specific to each applet in a package.
- Static resource - Includes options specific to each static resource in the CAP file.
- Sign - Includes options specific to the signing feature of the CAP file.

Table 5-3 JSON File Options for Converter

Option	Level	Description
`CAP_AID`	CAP file	The AID of the CAP file available as an executable load module.
`CAP_name`	CAP file	The name of the CAP file generated by the Converter. On the disk, the name of the CAP file would look like <*CAP_name*>`.cap` and all the components inside the CAP file will be located in the <*CAP_name*>`/javacard` directory.
`CAP_version`	CAP file	The user-defined version of the CAP file as an executable load module.
`debug`	CAP file	Generates the optional debug component of a CAP file. The same rules apply to the compact mode.
`noverify`	CAP file	Suppresses the verification of input and output files. The same rules apply to the compact mode.
`verbose`	CAP file	Enables verbose output.
`outputDir`	CAP file	Sets the root directory for output of the CAP file.
`nowarn`	CAP file	Instructs the Converter not to report warning messages.
`nobanner`	CAP file	Suppresses all banner messages.

Table 5-3 (Cont.) JSON File Options for Converter

Option	Level	Description
`useCapComponents`	CAP file	Instructs the Converter to retain the user-defined CAP components instead of generating them in the final CAP bundle. The input format is as follows: <*CAP_name*>`/javacard/*.cap`
`CAP`	CAP file	Instructs the Converter to write or not to write the CAP file to the disk.
`integer`	CAP file	Instructs the Converter to support the 32-bit integer type.
`exportPath`	CAP file	Specifies the root directories in which the Converter looks for the export files. The same rules apply for the compact mode. Note that the Java Card API framework export files directory is not required and `-traget 3.1.0` option is used automatically in the extended mode.
`inputPackages`	CAP file	An array of JSON objects, each representing the configuration for the Java package to be converted and added to the CAP file.
`staticResources`	CAP file	An array of JSON objects. Each representing the configuration for a static resource to be loaded on the CAP file.
`sign`	CAP file	A JSON object representing the configuration for signing the CAP file, which is generated by the Converter.
`PackageAID`	Package	The AID of the package as defined in the compact mode.
`PackageName`	Package	The fully-qualified name of the package as defined in the compact mode.
`baseDir`	Package	Sets the root directory from where the Converter looks for the classes in the package. If this option is not specified, the Converter uses the location of the configuration file as the root directory.
`outputDir`	Package	Sets the root directory for output of the JCA and EXP files generated for this package. The same rules apply for the compact mode. If this option is not set, the `baseDir` value is taken.
`public`	Package	Specifies if a package is exported or not. The values and its description follow: • If the value is set to `true`, and the package is a library, then all the public classes and interfaces are exported. • If the value is set to `true`, and the package is an applet package, then only shareable interfaces are exported. • If the value is set to `false`, nothing from the package is exported. Also, the AID field of the package will not appear in the header component of the CAP file and the AID field is ignored. Because of this, in the generated JCA files, the AID of a private package will have a random value. For a private package, the EXP file is not generated and the value of the EXP file is ignored.

Table 5-3   (Cont.) JSON File Options for Converter

Option	Level	Description
`version`	Package	The user-defined version of the package as defined in the compact mode. If the package is private and the `exportmap` field is set to `false`, the `version` field is ignored.
`JCA`	Package	Instructs the Converter to write or not to write the JCA file to the disk for the package.
`EXP`	Package	Instructs the Converter to write or not to write the EXP file to the disk for the package. If the package doesn't have an export component (if the package is private or an applet package with no shareable interfaces), the EXP file is not generated. Therefore, the EXP field is ignored.
`exportmap`	Package	Uses the token mapping from the predefined export file of the package. The Converter looks for the export file in the given `exportpath` at the CAP file level. If this field is set to `false` and the package is private, the `version` field is ignored.
`applets`	Package	An array of JSON objects. Each representing the configuration for a Java Card applet contained in this package.
`ClassAID`	Applet	Specifies the AID of the applet.
`ClassName`	Applet	Specifies the fully-qualified Java class name for this applet.
`id`	Static Resource	An integer representing the identification number for the static resource. The static resource IDs must be unique across the CAP file.
`file`	Static Resource	A valid system path to an existent and accessible file on the disk. The contents of this file is loaded as binary data in the CAP file for the static resource.
`keystore`	Sign	The keystore used in signing.
`storepass`	Sign	The keystore password.
`alias`	Sign	The keystore alias used in signing.
`passkey`	Sign	The alias password.

Handling Relative Paths

In the JSON configuration file, all the fields that have values for the paths to directories or files on disk, support relative paths.

These fields include: `outputDir`, `baseDir`, `exportPath`, and `file`. All the relative paths are defined relative to the directory in which the JSON configuration file is located.

For example, the static resources are defined as follows:

```
"staticResources":[{
            "id" : 1,
            "file" : "staticres\\static1.res"
        }, {
            "id" : 2,
            "file" : "staticres\\static2.res"
        }]
```

If the JSON configuration file is in the following location:

```
C:\Users\Test
```

Then the Converter finds the data for the static resources in the following locations:

```
C:\Users\Test\staticres\static1.res

C:\Users\Test\staticres\static2.res
```

This applies for any input or output relative path directory. In case of a list of paths, like the `exportPath` field, the preceding statements apply for each path in the list.

Converter JSON Configuration File Sample

The Converter JSON configuration file sample follows:

```
{
    "inputConfig": {
        "CAP_AID": "01:02:03:04:05:10",
        "CAP_name": "hellosample",
        "CAP_version": "1.0",
        "debug": true,
        "noverify": false,
        "verbose": true,
        "outputDir": "thecapfile",
        "exportPath": ".;.\\package1",
        "inputPackages": [{
            "baseDir": "package1",
            "PackageName": "com.lib",
            "PackageAID": "01:02:03:04:05:06",
            "public": true,
            "JCA": true,
            "EXP": true,
            "exportmap": true,
            "version": "1.1"
        }, {
            "PackageName": "com.mine",
            "baseDir": "package2",
            "public": false,
            "JCA": true,
            "EXP": false,
            "exportmap": false,
        }, {
            "PackageName": "com.sample",
            "PackageAID": "01:02:03:04:05:07",
            "baseDir": "package3",
            "public": true,
            "version": "1.0",
            "JCA": true,
            "EXP": true,
            "exportmap": false,
            "applets": [{
                "ClassAID": "01:02:03:04:05:07:01",
                "ClassName": "com.sample.MyApplet"
            }]
        }],
        "staticResources":[{
            "id" : 1,
            "file" : "staticres\\static1.res"
        }, {
            "id" : 2,
            "file" : "staticres\\static2.res"
        }]
    }
}
```

Validating a JSON Configuration File

To validate a JSON configuration file, a JSON schema file must be generated using the `-help JSON` option.

To save the schema file, use the following command for the Microsoft Windows operating system:

```
converter.bat -help JSON > converter_schema.json
```

The saved schema file can be used as an input to the validation tools for validating the actual JSON configuration files that are passed to the Converter. See `https://json-schema.org`, for more information on JSON schema and validation.

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 comply with the definitions in the Java Card Virtual Machine specification.

This section includes the following:

- Input File Naming Conventions
- Output File Naming Conventions
- Verification of Input and Output Files
- Creating a debug.msk Output File

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. In the compact mode, 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 the directory from which the user invoked the Converter. In the extended mode, the root directory can be set from the JSON configuration file using the `baseDir` field. This is set for each package contained in the extended CAP file. If the field is not specified for a specific package, the root directory for that package defaults to the directory in which the JSON configuration file resides.

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

`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.

If you use the `baseDir` field to specify the root directory as `C:\`*mywork*, the JSON field looks like this: `"baseDir":"C:\\`*mywork*"

The Converter looks for all class files in the `java.lang` package in the directory `C:\`*mywork*`\java\lang`.

Output File Naming Conventions

In the compact mode, the name of the CAP file, export file, and the Java Card Assembly file must be the last portion of the package name followed by the extensions`.cap`,`.exp`, and`.jca`, respectively. In the extended mode, the name of the CAP file is the value of the `CAP_name` field defined in the JSON configuration file followed by the `.cap` extension. For the export files and Java Card Assembly files generated in this mode, the same rules as in the compact mode apply.

By default, the files output from the Converter are written to a directory called `javacard`. This is a subdirectory of the input package's directory for the compact mode, or a subdirectory of the CAP name directory for the extended mode.

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

In the above `baseDir` example, assume that if the `CAP_name` field has the *"hellosample"* value, by default, the output files are written to the directory `C:\`*mywork*`\`*hellosample*`\javacard`.

The `-d` flag or the `outputDir` field enable you to specify a different root directory for the output.

In the above example, if you use the `-d` flag or the `outputDir` field to specify the root directory for the output to be`C:\`*myoutput*, the Converter writes the output files to the directory `C:\`*myoutput*`\java\lang\javacard` or `C:\`*myoutput*`\`*hellosample*`\javacard`, respectively.

When generating a CAP file, the Converter creates one or more Java Card Assembly files in the output directory as an intermediate result. If you don't want a Java Card Assembly file to be produced, omit the option `-out JCA` in the compact mode or set the JCA field to `false` for the respective package in the JSON configuration file in the extended mode. The Converter deletes the Java Card Assembly files 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 do not pass verification, then the output EXP and CAP files are deleted.

If you want to bypass verification of your input and output files, use the `-noverify` command line option or set the `noverify` field in the JSON configuration file to `true`. Note that if the Converter finds any errors, output files are not produced.

> **Note:**
>
> When using the Java Card off-card verifier to verify an extended CAP file, all EXP files that are required by the packages and are present inside the extended CAP file, must pass the verification.

Creating a debug.msk Output File

To create a `debug.msk` output file:

1. Set the `-mask` and `-debug` options described in Table 5-2 when you run the Converter.
2. Verify that the file `debug.msk` is created in the same directory as the other output files.

Using Export Files

A Java Card technology-based 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. You can use this package's export file 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.

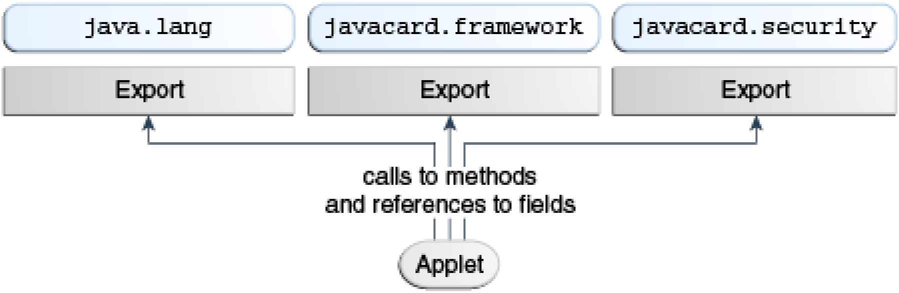
For more information on export files, see Verifying CAP and Export Files.

Figure 5-1 illustrates how an applet package is linked with the `java.lang`, the `javacard.framework` and `javacard.security` packages through their export files.

You can use the `-exportpath` command option and the `exportPath` JSON field 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 5-1 Calls Between Packages Go Through The Export Files



Specifying an Export Map

By specifying an export map, you can request the Converter to convert a package by using the tokens in the pre-defined export file of the package that is being converted. There are two distinct cases when using the `-exportmap` flag:

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

  During package reimplementation, the API of the package (exportable classes, interfaces, fields and methods) must remain 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 the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1*).

For example, if you have developed a package and would like to reimplement a method (package reimplementation) 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.

To specify an export map:

1. Set the `-exportmap` command option described in Table 5-2 when you run the Converter in the compact mode..

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

2. Set the `exportmap` JSON field to `true` for each package from an extended CAP file, for which you want to preserve binary compatibility, in the extended mode.

Viewing an Export File as Text

The `exp2text` tool is provided to allow you to view any export file in text format. The file to invoke `exp2text` is a batch file (`exp2text.bat`) that must be run from a working directory of *JC_HOME_TOOLS*`\bin` in order for the code to execute properly.

To view an export file as text:

- Enter the following command (see Table 5-4 for a description of the options):

`exp2text.bat` [*options*] *package-name*

Table 5-4   exp2text Command Line Options

Option	Description
`-classdir` *input-root-directory*	Specifies the root directory where the program looks for the export file.
`-d` *output-root-directory*	Specifies the root directory for output.
`-help`	Prints help message.

6

Working With CAP Files

This chapter 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.
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 describe a Java language package. In addition to the components, the CAP file also contains the manifest file `META-INF/MANIFEST.MF`, which you can use to improve distribution.

This chapter contains the following sections:

- Compact CAP File and Manifest File Syntax
- Extended CAP File Manifest File Syntax
- Generating a CAP File From a Java Card Assembly File
- Producing a Text Representation of a CAP File

Compact CAP File and 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. You can use this information to facilitate the distribution and processing of the CAP file.

The following information applies to the version 2.3 compact CAP files generated with the version 3.1.0 Converter and version 2.2 or 2.1 CAP files generated with the version 3.1.0 or later Converter.

The information in the manifest file is presented in name:value pairs. These name:value pairs are described in Table 6-1.

Table 6-1 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. Default: `3.1.0`.
`Java-Card-Converter-Provider`	Provider of the converter tool. For example: `Oracle Corporation`

Table 6-1 (Cont.) Name:Value Pairs in the MANIFEST.MF File

Name	Value
`Java-Card-CAP-File-Version`	CAP file *major.minor* version. Possible values are: `2.1`, `2.2`, or `2.3`.
`Java-Card-Package-Version`	The *major.minor* 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 *n*. For example: `0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x05`
`Java-Card-Applet-<n>-Name`	Simple class name for applet *n*. For example: `MyApplet`
`Java-Card-Import-Package-<n>-AID`	The AID for imported package *n*. For example: `0xa0:0x00:0x00:0x00:0x62:0x00:0x01`
`Java-Card-Import-Package-<n>-Version`	The *major.minor* version of imported package *n*. 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 (Oracle Corporation)
Java-Card-CAP-Creation-Time: Tue Jan 15 11:07:55 PST 2010
Java-Card-Converter-Version: 1.3
Java-Card-Converter-Provider: Oracle Corporation
```

```
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
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
```

Extended CAP File Manifest File Syntax

An extended CAP file utilizes the JAR file format, and has the same properties as a compact CAP file. However, there are some differences in the way the information that is specific to the extended CAP file is represented in the `META-INF/MANIFEST.MF` file.

The following table lists the names in the manifest file that are specific to the Java Card packages and Java Card Applets. These fields are changed to consider the extended CAP file context:

The information in the manifest file is presented in name:value pairs. These name:value pairs are described in Sample Extended CAP Manifest File.

Table 6-2 Extended CAP File Manifest File Name Syntax

Name	Changed To:	Change Description
`Java-Card-Package-AID`	`Java-Card-Package-<n>-AID`	An extended CAP file can have multiple packages. Therefore, an index is added for each package name.
`Java-Card-Package-Version`	`Java-Card-Package-<n>-Version`	An extended CAP file can have multiple packages. Therefore, an index is added for each package version. If the package is not exported (private or applet package with no shareable interfaces), the value of this field is set to `private`.
`Java-Card-Applet-<n>-AID`	`Java-Card-Package-<m>-Java-Card-Applet-<n>-AID`	An extended CAP file can have multiple packages. Therefore, the package that contains the applet is added for each applet AID.
`Java-Card-Applet-<n>-Name`	`Java-Card-Package-<m>-Java-Card-Applet-<n>-Name`	An extended CAP file can have multiple packages. Therefore, the package that contains the applet is added for each applet name.

Some new name:value pairs are added in the extended CAP manifest file. These name value pairs have extended CAP file-specific information. The following table lists and describes the new name value pairs.

Table 6-3 Name:Value Pairs in the extended CAP MANIFEST.MF File

Name	Value
`Java-Card-CAP-Name`	The extended CAP file name as defined in the `CAP_name` field from the JSON input configuration file.
`Java-Card-CAP-AID`	The extended CAP file AID as present in the header component of the CAP file.
`Java-Card-CAP-Version`	The extended CAP file version as present in the header component of the CAP file.

Sample Extended CAP Manifest File

The following code sample illustrates the sample extended CAP `MANIFEST.MF` file.

```
Manifest-Version: 1.0
Created-By: 1.7.0_60 (Oracle Corporation)
Name: BigApplet007
Java-Card-Integer-Support-Required: FALSE
Java-Card-Imported-Package-1-AID: 0xa0:0x00:0x00:0x00:0x62:0x00:0x01
Java-Card-Package-1-Name: com.oracle.lib
Java-Card-CAP-Version: 1.0
Java-Card-Package-3-Java-Card-Applet-1-AID: 0x01:0x02:0x03:0x04:0x05:0x06:0x01
Java-Card-Imported-Package-1-Version: 1.0
Java-Card-Package-3-Java-Card-Applet-1-Name: BigApplet001
Java-Card-Package-4-AID: private
Java-Card-CAP-Creation-Time: Thu Dec 06 18:47:17 FET 2018
Java-Card-Converter-Provider: Oracle Corporation
Java-Card-Package-4-Version: private
Java-Card-Package-2-Name: com.oracle.ext
Java-Card-Package-1-AID: 0x01:0x02:0x03:0x04:0x05:0x09
Java-Card-Package-4-Java-Card-Applet-2-Name: BigApplet001
Java-Card-Package-3-Name: com.oracle.bigapp
Java-Card-Package-3-Version: private
Java-Card-CAP-Name: BigApplet007
Java-Card-Package-2-Version: 1.0
Java-Card-Converter-Version:  [v3.1.0]
Java-Card-Package-4-Java-Card-Applet-1-Name: BigApplet002
Java-Card-Imported-Package-2-AID: 0xa0:0x00:0x00:0x00:0x62:0x01:0x01
Java-Card-Package-2-AID: 0x01:0x02:0x03:0x04:0x05:0x0b
Java-Card-Package-4-Java-Card-Applet-1-AID: 0x01:0x02:0x03:0x04:0x05:0x07:0x01
Java-Card-Package-4-Java-Card-Applet-2-AID: 0x01:0x02:0x03:0x04:0x05:0x08:0x01
Java-Card-CAP-AID: 0x01:0x02:0x03:0x04:0x05:0x06:0x0a
Java-Card-Package-4-Name: com.oracle.bigapp02
Java-Card-CAP-File-Version: 2.3
Java-Card-Package-3-AID: private
Java-Card-Imported-Package-2-Version: 1.7
Java-Card-Package-1-Version: 1.0
```

Generating CAP Files From Java Card Assembly Files

Use the `capgen` tool to generate a compact CAP file from a given Java Card Assembly file or an extended CAP file from one or more Java Card Assembly files. 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

To run `capgen`:

- Enter the following on the command line (see Table 6-4 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_HOME_TOOLS*`\bin` in order for the code to execute properly.

Table 6-4 capgen Command Line Options

Option	Description
`-help`	Prints a help message.
`-nobanner`	Suppresses all banner messages.
filename	Specifies the Java Card Assembly file in case of the compact CAP file generation or a capgen JSON configuration file in case of the extended CAP file generation.
`-o` *filename*	Enables you to specify an output file. If the output file is not specified with the `-o` flag, output defaults to the file `a.jar` in the current directory.
`-version`	Outputs the version information.
`-config`	Enables capgen to run in the extended mode. In this case, the *filename* parameter is a JSON configuration file, similar to the one given for the Converter in the extended mode. The JCA input files are defined in the configuration file.

Using a JSON Configuration File for capgen in the Extended Mode

In the extended mode, the capgen tool generates extended CAP files, from one or multiple Java Card Assembly files.

For using the capgen tool in the extended mode, a JSON configuration file must be used with the `-config` option. This JSON file is similar to the one used by the Converter tool (see Using a JSON Configuration File for Converter in the Extended Mode). The only difference is, some of the general conversion parameters that are used by the Converter tool, including the export paths, are not used by the capgen tool. This is because, this information is already present in the JCA files. For each of the package, only the path to the JCA files is provided. The information that is not present in the JCA files remain in the JSON file, similar to the extended CAP file information and static resources information.

The configuration in the JSON file is a JSON object with a single field, `inputConfig`. All other fields are defined inside this field at different levels of hierarchy. The description of levels follows:

Table 6-5 JSON File Options for capgen

Option	Level	Description
`CAP_AID`	CAP file	The AID of the CAP file available as an executable load module.
`CAP_name`	CAP file	The name of the CAP file generated by the Converter. On the disk, the name of the CAP file would look like <*CAP_name*>`.cap` and all the components inside the CAP file will be located in the <*CAP_name*>`/javacard` directory.
`CAP_version`	CAP file	The user-defined version of the CAP file as an executable load module.
`debug`	CAP file	Generates the optional debug component of a CAP file. The same rules apply to the compact mode.
`outputDir`	CAP file	Sets the root directory for output of the CAP file.
`inputPackages`	CAP file	An array of JSON objects, each representing the configuration for the Java package to be converted and added to the CAP file.
`staticResources`	CAP file	An array of JSON objects. Each representing the configuration for a static resource to be loaded on the CAP file.
`jcainputfile`	Package	A valid path to an existent and accessible Java Card Assembly file converted for this package. This path can be given as a relative path. Relative paths conform to the same rules as the Converter JSON configuration files. These are relative to the location of the JSON configuration file.
`outputDir`	Package	Sets the root directory for output of the JCA and EXP files generated for this package. The same rules apply for the compact mode. If this option is not set, the `baseDir` value is taken.
`id`	Static Resource	An integer representing the identification number for the static resource. The static resource IDs must be unique across the CAP file.
`file`	Static Resource	A valid system path to an existent and accessible file on the disk. The contents of this file is loaded as binary data in the CAP file for the static resource.

Capgen JSON Configuration File Sample

The capgen JSON configuration file sample follows:

```
{
    "inputConfig": {
        "CAP_AID": "01:02:03:04:05:10",
        "CAP_name": "hellosample",
        "CAP_version": "1.0",
        "debug": true,
        "outputDir": "thecapfile",
        "inputPackages": [{
            "jcainputfile": "package1\\com\\lib\\javacard\\lib.jca"
        }, {
            "jcainputfile": "package2\\com\\mine\\javacard\\mine.jca"
        }, {
            "jcainputfile": "package3\\com\\sample\\javacard\\sample.jca",
        }],
        "staticResources":[{
            "id" : 1,
            "file" : "staticres\\static1.res"
        }, {
            "id" : 2,
            "file" : "staticres\\static2.res"
        }]
    }
}
```

Producing a Text Representation of a CAP File

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

Running capdump

To run `capdump`:

- Enter the following on the command line:

  `capdump.bat` *filename*

  There are no command line options, *filename* is the CAP file, and output from the command is always written to standard output.

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

7

Debugging Applications

This chapter describes the debug proxy tool that is included in the development kit. You can use it either within the Eclipse IDE or as a separate tool with any Java IDE. This chapter contains the following sections:

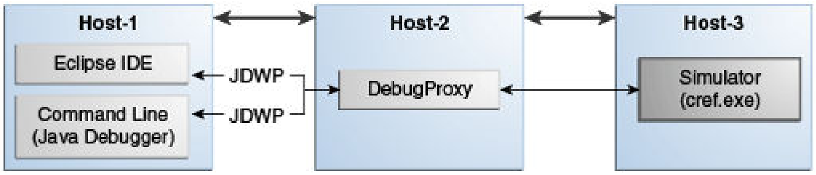
- Debugger Architecture
- Running the Debug Proxy From the Command Line

Debugger Architecture

You can use `cref`, `jc-debug-proxy`, and an IDE to debug your project.

The prebuilt executable runtime environment, `cref` is run from inside Eclipse or on the command line, and has the ability to simulate persistent memory (EEPROM) and to save and restore the contents of EEPROM to and from disk files. It performs I/O through a socket interface, simulating the interaction between a card reader and a host computer.

Figure 7-1 Debugger Architecture



The Java Debug Wire Protocol (JDWP) used by the IDE is heavy for a small VM such as that provided by the simulator. Instead, the simulator uses a lightweight proprietary protocol to provide a minimum set of debugging capabilities. The debugger tool, `jc-debug-proxy`, translates commands and responses between `cref` and the IDE into the appropriate protocol.

Because `cref`, `jc-debug-proxy`, and the IDE communicate through sockets, you may debug using a remote host. For example, `cref` could run on host1, `jc-debug-proxy` could run on host2, and the IDE could run on host3.

Ports used between the IDE and `jc-debug-proxy`, and `jc-debug-proxy` and `cref`, are distinguished by the names "listen port" and "remote port".

Running the Debug Proxy from the Command Line

If you are not using Eclipse for development, you can run the debug proxy and attach another Java technology-enabled debugger to it from the command line.

To run the debugger:

1. Compile the application's class files using the `-g` option. If the -g option is not used, it is not possible to set breakpoints in the source code
2. Generate APDU scripts for applet installation, instance creation and selection by using the script generator tool (`scriptgen.bat`).
3. Start `cref` in debug mode.

   You must set the `-debugPort` option so that `cref` opens the specified port to communication with debug proxy. Without this option, the debugging functionality in `cref` is disabled.

   For example:

   ```
   JC_HOME_SIMULATOR\bin\cref_tdual.exe -debugPort 9090[options]
   ```

4. Run the APDU scripts.

   APDU scripts can be executed using `apdutool.bat`. At a minimum, the installation script must be executed before the debug proxy connects to the VM. Other scripts can be executed later to debug the applet's `install()` and `process()` methods
5. Start `jc-debug-proxy` as described in Starting the Debugger.

   For example:

   ```
     java.exe -jar lib\jc-debug-proxy.jar -capPath
   C:\workspace\HelloWorld\deliverables\hello\javacard\hello.cap -vmPort
   9090   -port 8000
   ```

6. Attach the debugger to the debug proxy.

   NetBeans or any other Java-compatible debugger can be used to connect to the debug proxy using the JDWP protocol. The debugger needs to be configured to connect to the remote Java application running on a specific host and port.

For an example, see:

Debugging the HelloWorld Sample from the Command Line

Debug Proxy Options

To run the debug proxy from the command line, use the following command syntax:

```
java.exe –jar lib\jc-debug-proxy.jar <debug proxy arguments>
```

The command line arguments for the debug proxy are:

Command Line Argument	Description
`-debug-info`	The source debug-info file that contains debug information for system classes
`-gen-debug-info`	Starts debug proxy in *generate debug-info mode* to generate the system classes debug information file using `.exp` files found on the provided path
`-port`	The port that the Java debugger connects to
`-vmPort`	The port that the VM listens on.
`-vmHost`	The hostname of the system the VM is running on
`-capPath`	Required. Path to the cap file(s) being debugged.
`-help`	Short description of help

For example:

```
java.exe -jar lib\jc-debug-proxy.jar -capPath
C:\workspace\HelloWorld\deliverables\hello\javacard\hello.cap -vmPort 9090
-port 8000
```

Debugging the HelloWorld Sample from the Command Line

To debug the `HelloWorld` sample from the command line:

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

      ```
      cref –o hello.eeprom
      ```

      > **Note:**
      >
      > The `-o` command line option instructs `cref` to save the EEPROM data to the `hello.eeprom` file before terminating.

2. Open a second Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\classic_applets\HelloWorld\applet` directory.
   b. Open the `applet.opt` file in a text editor and add a new line with `-debug` option. This option will be passed to the converter to generate debug information.
   c. At the command prompt, invoke `ant` with a target set to `all`. The output file is `default.out` or, optionally, you can specify a different output file with the `-D` parameter:

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

      This builds the applet, executes the APDU script, and creates an output file in the applet directory.

3. `cref` terminates. Restart it in the first window by entering this command:

   ```
   cref –debugPort 9090 –i hello.eeprom
   ```

4. In the second command prompt, navigate to the `JC_HOME_SIMULATOR\lib` directory and start the debug proxy:

   ```
   java.exe -jar jc-debug-proxy.jar -capPath
   JC_HOME_SIMULATOR\samples\classic_applets\HelloWorld\applet\build\clas
   ses\com\sun\jcclassic\samples\helloworld\javacard\helloworld.cap
   ```

5. Start the Java debugger of your choice and attach it to the 8000 port of the local host.
6. Now you can set a break point and see it hit after a proper APDU is issued using the `apdutool`.

8

Packaging and Deploying Your Application

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

Overview of Packaging and Deploying Applications

You can use the development kit installer to:

- Download a Java Card technology CAP file to a Java Card technology-compliant smart card, or during development, to the Java Card RE.
- 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 smart card in the form of APDU commands. See Using the On-card Installer for Deletion for more information.
- Set default applets on different logical channels.

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 on-card installer is not a multi selectable 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`.

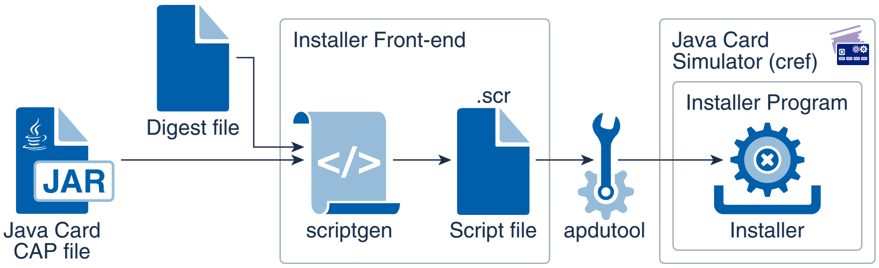
Installer Components and Data Flow

The following illustration shows the components of the installer and how they interact with the other parts of Java Card technology.

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

For more information about the installer, see the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.1*.

Figure 8-1   Installer components



The data flow of the installation process is as follows:

1. The `scriptgen` takes the CAP file produced by the Converter and generates a script file that contains a sequence of APDU commands.
2. This set of APDUs is read by `apdutool` tool and sent to the on-card installer.
3. The on-card installer processes the CAP file contents contained in the APDU commands, and sends a response APDU containing a status and, optionally, the 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 *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1*. If you have a source release, you can localize data associated with the `scriptgen` tool.

- Enter the following command to run the `scriptgen` tool :

  ```
  scriptgen.bat [options] -hashfile hash-file-path cap-file-path
  ```

  The `scriptgen.bat` file is used to invoke `scriptgen` and must be run from a working directory of *JC_HOME_SIMULATOR*`\bin` in order for the code to execute properly. Table 8-1 describes the options that can be used to invoke `scriptgen`.

Table 8-1   scriptgen Command Line Options

Option	Description
`-help`	Prints a help message and exits.
cap-file-path	CAP file name including the full absolute path.
`-hashfile` *hash-file-path*	Fully qualified path and name of the verifier-generated file that contains the hashes for all of the components in the input CAP file.
`-nobanner`	Suppresses printing of the banner.
`-nobeginend`	Suppresses the output of the CAP Begin and CAP End APDU commands.
`-o` *filename*	Output filename (default is `stdout`).

Table 8-1 (Cont.) scriptgen Command Line Options

Option	Description
`-package` *package-name*	The name of the package contained in the CAP file. If the CAP file contains components of multiple packages, you must use this option to specify which package to process.
`-version`	Prints the version number and exits.

> **Note:**
>
> The `apdutool` commands of `powerup;` and `powerdown;` are not included in the output from `scriptgen`.

Sending and Receiving APDUs

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

This section includes the following topics:

- Running apdutool
- apdutool Examples
- Using APDU Script Files
- Setting Default Applets
- On-Card Installer Applet AID

Running apdutool

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

To run the `apdutool`:

- Enter the following command (see Table 8-2 for a description of the options):

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

The `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 8-2 apdutool Command Line Options

Option	Description
`-help`	Displays online help for the command.
`-h` *host-name*	Specifies the host name on which the TCP/IP socket port is found. (See the flag `-p`.)

`-d`
or
`-descriptiveoutput`
	Formats the output in more user-readable form.

`-k`	When using preprocessor directives in an APDU script, this option generates a related preprocessed APDU script file in the same directory as the APDU script.
`-noatr`	Suppresses outputting an ATR (answer to reset).
`-nobanner`	Suppresses all banner messages.
`-o` *output-file*	Specifies an output file. If an output file is not specified with the `-o` flag, output defaults to standard output.
`-p` *port-number*	Specifies a TCP/IP socket port other than the default port (which is 9025).
`-t0`	Runs T=0 single interface.
`-verbose`	If enabled, enables verbose apdutool output.
`-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.
input-file-name	Specifies an input script file.

apdutool Examples

The following examples show how to use `apdutool` in:

- Directing Output to the Console
- Directing Output to a File

Directing Output to the Console

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

To direct output to the console:

- Enter the following command:

```
apdutool example.scr
```

Directing Output to a File

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

To direct output to a file:

- Enter the following command:

  ```
  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 8-3 describes each supported script file command in detail noting that they are not case sensitive.

> **Note:**
>
> All APDU script file commands are not case-sensitive.

Table 8-3 Supported APDU Script File Commands

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

Table 8-3 (Cont.) Supported APDU Script File Commands

Command	Description
`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` *AID*;	Selects the applet with the specified AID, where *AID* identifies the applet to be selected in the form of `//aid/A005453412/151146712`. For example: `select //aid/A000000062/03010C0101;`
`open channel` [*channel-no*] [`on` *origin-channel*];	Opens the channel with the channel number specified by *channel-no* on the origin channel specified by *origin-channel*, where *channel-no* is an integer. The default value for the origin channel is basic channel number 0. *channel-no* and *origin-channel* are both optional. *origin-channel* must be an integer from 0-19.
`close channel` *channel-no* [`on` *origin-channel*];	Closes the channel having the channel number specified by *channel-no* on origin channel *origin-channel*, where *channel-no* is an integer. `on` *origin-channel* is optional and the default value for *origin-channel* is basic channel number `0`. *origin-channel* must be an integer from 0-19.
`send` *APDU* [`to` *AID*] [`on` *origin-channel*];	Sends the APDU specified by *APDU* after selecting the applet specified by *AID* on the specified origin channel, where the *APDU* format uses the C-APDU syntax of the `apdutool`. `on` *origin-channel* 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` *AID* 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 0x080 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 simulator supports setting distinct default applets on distinct logical channels and distinct interfaces. You can use this request 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 simulator prior to issuing this command.

Table 8-4 Set Default Applets on Different Logical Channels

Applet AID	Lc Field	Data	Le Field
`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 is configured as the 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:

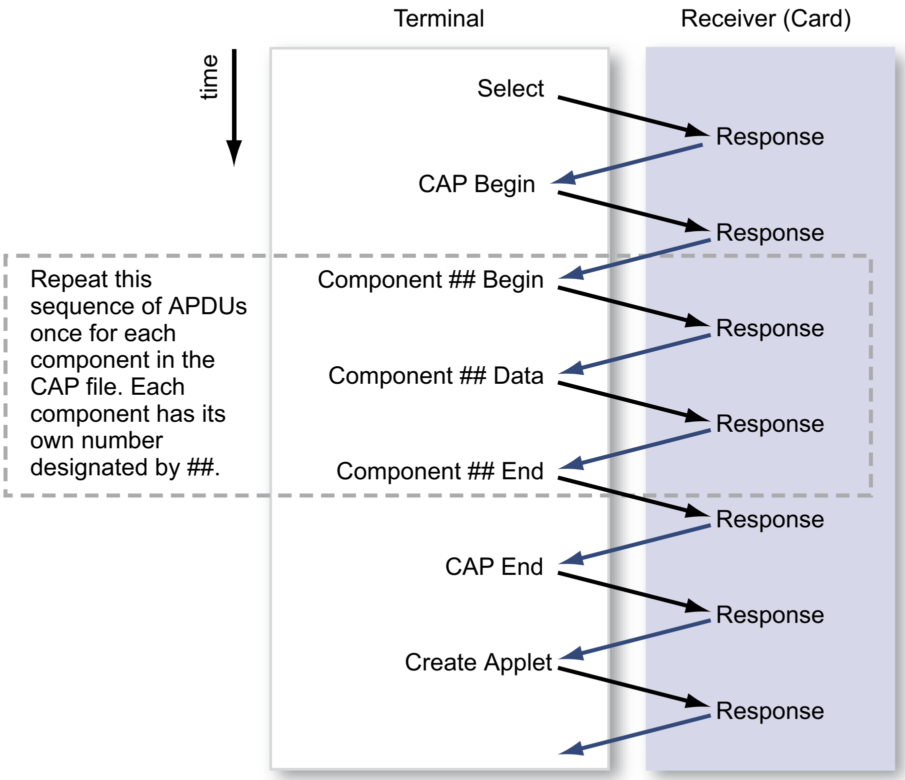
```
powerup;
// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08
    0x01 0x7F;
// create JavaPurse
0x80 0xB8 0x00 0x00 0x0b 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01
    0x04 0x01 0x00
0x7F;
powerdown;
```

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 8-2.

Figure 8-2 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 APDU Command
- Response APDU Command
- 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 APDU Command

Table 8-5 specifies the field sequence in the `Select` APDU, which is used to invoke the on-card installer.

Table 8-5 Select APDU Command

Command	Lc	Installer	Le
`0x0x, 0xa4, 0x04, 0x00`	Lc field	Installer AID	Le field

Response APDU Command

Table 8-6 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 8-6 Response APDU Command

Data	Response
[optional response data]	SW1SW2

CAP Begin

Table 8-7 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 8-7 CAP Begin APDU Command

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

CAP End

Table 8-8 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 8-8 CAP End APDU Command

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

Component ## Begin

Table 8-9 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, and so on. 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 8-9 Component ## Begin APDU Command

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

Component ## End

Table 8-10 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 8-10   Component ## End APDU Command

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

Component ## Data

Table 8-11 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 8-11   Component ## Data APDU Command

Command	Lc	data	Le
`0x8x, 0xb4, 0x##, 0x00`	Lc field	Data field	Le field

Create Applet

Table 8-12 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 8-12   Create Applet APDU Command

Command	Lc	AID length	AID	Parameter length	Parameter	Le
`0x8x, 0xb8, 0x00, 0x00`	Lc field	AID length field	AID field	parameter length field	[parameters]	Le field

Abort

Table 8-13 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 8-13   Abort APDU Command

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

APDU Responses to Installation Requests

If a command completes successfully, the installer sends a response code of 0x9000. A number of codes can be sent in response to unsuccessful installation requests, as shown in Table 8-14.

Table 8-14 APDU Responses to Installation Requests

Response Code	Description
`0x6402`	Invalid CAP file magic number. • **Cause:** An incorrect magic number was specified in the CAP file. • **Solution:** Refer to the *Java Virtual Machine Specification* for the correct magic number. Ensure that the CAP file is built correctly, run it through `scriptgen`, and download the resulting script file to the card.
`0x6403`	Invalid CAP file minor number. • **Cause:** An invalid CAP file minor number was specified in the CAP file. • **Solution**: Refer to the *Java Virtual Machine Specification* for the correct minor number. Ensure that the CAP file is built correctly, run it through `scriptgen`, and download the resulting script file to the card.
`0x6404`	Invalid CAP file major number. • **Cause:** An invalid CAP file major number was specified in the CAP file. • **Solution**: Refer to the *Java Virtual Machine Specification* for the correct major number. Ensure that the CAP file is built correctly, run it through `scriptgen`, and download the resulting script file to the card.
`0x640b`	Integer not supported. • **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. • **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. • **Cause:** A duplicate Applet AID was detected in CREF. • **Solution:** Choose a new AID for the applet to be installed.
`0x640f`	Installation aborted. • **Cause:** Installation was aborted by an outside command. • **Solution:** Restart the CAP installation from the beginning and check the `INS` bytes in the installation script for the offending command.
`0x6421`	Installer in error state. • **Cause:** A non-recoverable error previously occurred. • **Solution:** Scan the `apdutool` output for previous APDU responses indicating an error. Restart the CAP installation.
`0x6422`	CAP file component out of order. • **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`	Exception occurred. • **Cause:** General purpose error in the installer or applet code. • **Solution:** Check your applet code for errors.

Table 8-14 (Cont.) APDU Responses to Installation Requests

Response Code	Description
`0x6425`	Install APDU command out of order. • **Cause:** Installer APDU commands were received out of order. • **Solution:** Check the script file for the order of APDU commands. See Sending and Receiving APDUs for more information on the ordering of APDU commands.
`0x6428`	Invalid component tag number. • **Cause:** An incorrect component tag number was detected during download. • **Solution:** Refer to Chapter 6 in the *Java Virtual Machine Specification* for the correct tag number.
`0x6436`	Invalid install instruction. • **Cause:** An invalid Installer APDU command was received. • **Solution:** Check the script file for the offending command. See Sending and Receiving APDUs for more information on APDU commands.
`0x6437`	On-card package max exceeded. • **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`	Imported package not found. • **Cause:** A package that is required by the current package was not found. • **Solution:** Download the required package first.
`0x643a`	On-card applet package max exceeded. • **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. • **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. • **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. • **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 `install` method, and so on.
`0x644f`	Package name is too long. • **Cause:** The package name exceeds the length specified in the *Java Virtual Machine Specification*. • **Solution:** Replace the name and rebuild.

Table 8-14 (Cont.) APDU Responses to Installation Requests

Response Code	Description
`0x6445`	Maximum allowable applet instances exceeded. • **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. • **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. • **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 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 the simulator provides the ability to delete CAP file 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 ADPU commands. Requests to delete an applet or CAP file cannot be sent from an applet on the card. For more information on CAP file and applet deletion, see the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.1*.

How to Send a Deletion Request

1. Select the on-card installer applet on the card.
2. Send the ADPU for the appropriate deletion request to the installer. The requests that you can send are described in the following sections:
   a. Delete CAP File

b. Delete CAP File and Applets

c. Delete Applets

For information on the responses that the ADPU requests can return, see APDU Responses to Deletion Requests.

APDU Requests to Delete CAP Files and Applets

You can send requests to delete a CAP file, a CAP file 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 CAP File

In this request, the Data field contains the size of the CAP file AID and the AID of the CAP file to be deleted. Table 8-15 shows the format of the Delete CAP File request and the expected response.

Table 8-15 Delete CAP File Command

Command	Lc	data	Le
`0x8x, 0xc0, 0x, 0x`*`XXXX`*	Lc field	Data field	Le field

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

```
//Delete CAP File 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 CAP file AID.

Delete CAP File and Applets

This request is similar to the Delete CAP file command. In this case the CAP file and applets are removed simultaneously. The data field contains the size of the CAP file AID and the AID of the CAP file to be deleted. Table 8-16 shows the format of the Delete CAP File and Applets request and the expected response.

Table 8-16 Delete CAP File and Applets Command

Command	Lc	data	Le
0x8x, 0xc2, 0x*XX*, 0x*XX*	Lc field	Data field	Le field

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

```
//Delete CAP file 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 CAP file 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 8-17 shows the format of the Delete Applet request and the expected response.

Table 8-17 Delete Applet Command

Command	Lc	data	Le
`0x8x, 0xc4, 0x0#, 0x`*XX*	Lc field	Data field	Le field

The value of 0xXX can be any value for the `P2` parameter. The installer ignores 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 8-18.

Table 8-18 APDU Responses to Deletion Requests

Response Code	Description
`0x6a86`	Invalid value for P1 or P2 parameter. • **Cause:** Value for P1 is less than 1 or greater than 8. • **Solution:** Ensure that the value for P1 is between 1 and 8.

Table 8-18 (Cont.) APDU Responses to Deletion Requests

Response Code	Description
`0x6443`	Applet not found for deletion. • **Cause:** The applet with the specified AID does not exist. • **Solution:** Check and correct the AID.
`0x644b`	Package not found. • **Cause:** The package with the specified AID does not exist. • **Solution:** Check and correct the AID.
`0x644c`	Dependencies on package. • **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`	One or more applet instances of this package are present. • **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`	Package is ROM package. • **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`	Dependencies on applet. • **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.
`0x6449`	Internal memory constraints. • **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.
`0x6451`	Cannot delete applet; the applet is currently active on one of the logical channels. • **Cause:** An attempt was made to delete an applet which is currently active on one of the logical channels. • **Solution:** Make sure that the applet is not selected on any of the logical channels. Then, re-attempt to delete the applet.
`0x6700`	Invalid value for `Lc` parameter. • **Cause:** In case of package deletion, the value for `Lc` is less than 6 or greater than 17. In case of applet deletion, the value for `Lc` is less than 7 or greater than 136. • **Solution:** Value of `Lc` in both of these cases depends on the AIDs being passed in the APDU. Make sure the AIDs are correct and value for `Lc` 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 8-19.

Table 8-19 APDU Response Format

data	Response
[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 CAP files to be downloaded is 32, including up to 16 CAP files with applets.
- 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 continue to fail after an error occurs.
- The maximum number of applets that can be deleted using one command is eight.

9

Verifying CAP and Export Files

This chapter describes off-card verification as a means for evaluating CAP and export files in a desktop environment. When applied to the set of CAP files that reside on a Java Card technology compliant secure element 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 secure element has been verified. Oracle's Off-Card Verifier supports incremental verification and resolution of the set of CAP files that are installed on a Java Card technology-compliant device in a desktop environment. The unit of verification is a single CAP file. The context in which a CAP file can be executed is provided through the Application Programming Interface (API) of referenced packages as defined in their export files. Resolution is validated off-card by examining the export files of referenced packages.

Oracle's Off-Card Verifier uses a bottom-up approach to verify the CAP files. In a nutshell, once a CAP file and its corresponding export file, if any, have been verified, it is not examined the succeeding times it is referenced. This is analogous to the process performed by an optimized Java virtual machine where, once the `java.lang` package has been loaded, verified, resolved, and initialized, it is not examined the succeeding times it is referenced. The same is true for a Java Card technology-compliant device.

A Java Card technology-enabled device is a secure environment. Additional security measures, such as the firewall, prevent a library from being corrupted. Once a verified CAP file has been installed on a Java Card technology-compliant device its state cannot be changed. This includes both its internal state and its context.

Off-Card verification provides a complete solution for Java Card technology-based applications when additional security constructs are applied. For more information on security measures and other details on working of the Off-Card Verifier, refer to the Off-card Verifier White paper.

This chapter contains the following sections:

- Overview of Verifying CAP and Export Files
- Verifying CAP Files
- Verifying Export Files
- Verifying Binary Compatibility
- Command Line Options for Off-Card Verifier Tools

Overview of Verifying CAP and Export Files

The off-card verifier is a combination of three tools, `verifycap`, `verifyexp`, and `verifyrev`. The following sections describe how to use each tool.

- `verifycap` - Verifying CAP Files
- `verifyexp` - Verifying Export Files

- verifyrev - Verifying Binary Compatibility

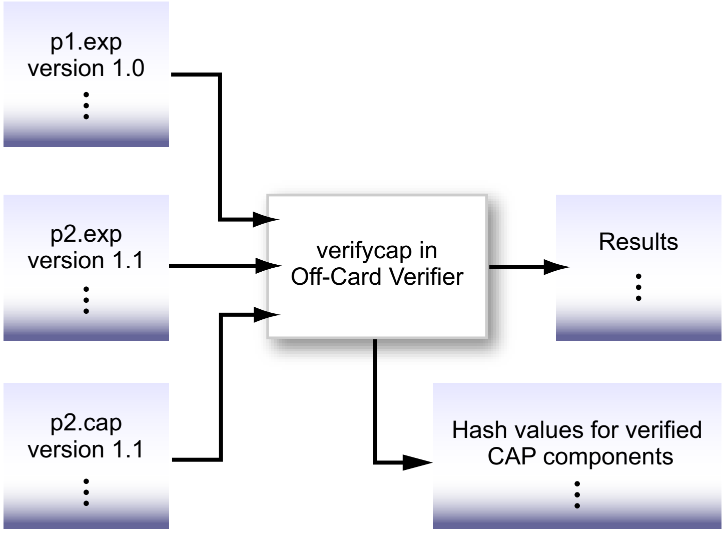
Verifying CAP Files

The verifycap tool is used to verify a CAP file within the context of packages' export files (if any) and the export files of imported packages. This verification confirms whether a CAP file is internally consistent, as defined in the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1*, and consistent with a context in which it can reside in a Java Card technology-enabled device.

To ensure the integrity of the CAP file to be downloaded on a card, the verifier computes and outputs hash values for each of the required CAP file components. To output the hash values in a text file, specify the command line parameter -outfile *hash-file-path*. If the -outfile parameter is not specified, the verifier outputs the hash values on the console output. A CAP file loader should compute the hash values for each of the required CAP components and verify them against the hash values produced by the verifier to assert the integrity of the CAP file being loaded on the card. The scriptgen tool in the Java Card Development kit performs the hash computation and comparison before generating the download script for a CAP file. For more information about the scriptgen tool, see Running scriptgen.

Each individual export file is verified as a single unit. The scenario is shown in Figure 9-1. 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 9-1 Verifying a CAP file



Running verifycap

The file to invoke verifycap is a batch file (verifycap.bat) that you must run from a working directory of *JC_HOME_TOOLS*\bin in order for the code to execute properly.

To run verifycap:

- Enter the following command (Table 9-1 describes the available options):

  `verifycap.bat` [*options*] *export-files CAP-file*

Table 9-1 verifycap Command Line Arguments

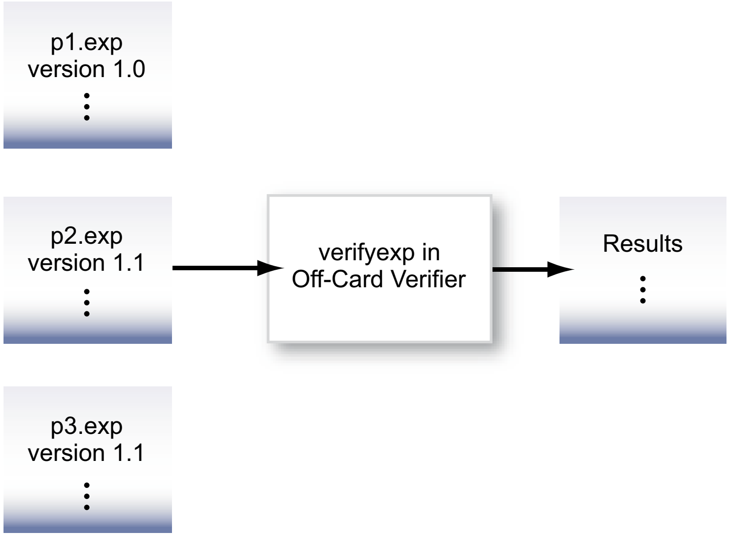
Argument	Description
export-files	A list of export files of the packages that this CAP file uses could be either one of the following: • Export files corresponding to the package version available on the target platform. • Export files corresponding to the version of imported packages. In this case, you also need to check that these export files are binary compatible with export files corresponding to the packages available on the target platform. Note that, when using this option in conjunction with the `-target` command line argument, any export file in this list corresponding to a Java Card platform package will automatically be overridden by the `verifier` to use an internal copy of the export file matching the specified target version. For more information, see the `-target` command line argument in Table 9-3.
CAP-files	Name of the CAP file to be verified.
`-digest` *`digest-algorithm-name`*	Specifies the digest algorithm to use for computing hash values for required CAP components. If this option is not specified or an invalid algorithm name is specified, the verifier uses SHA-256 as the default algorithm.
`-outfile` *`hash-output-file-path`*	Specifies the path to the text file that the verifier uses to output the computed hash values for the required CAP components. If this option is not specified, hash values are output to the system console.

Command Line Options for Off-Card Verifier Tools describes additional `verifycap` options.

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 the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1*. This scenario is illustrated in Figure 9-2.

Figure 9-2 Verifying An Export File



Running verifyexp

The file that invokes `verifyexp` is a batch file (`verifyexp.bat`) that you must run from a working directory of *JC_HOME_TOOLS*`\bin` for the code to execute properly.

To run `verifyexp`:

- Enter the following command (Table 9-2 describes the available options):

  `verifyexp` [*options*] *export-file*

Table 9-2 verifyexp Command Line Argument

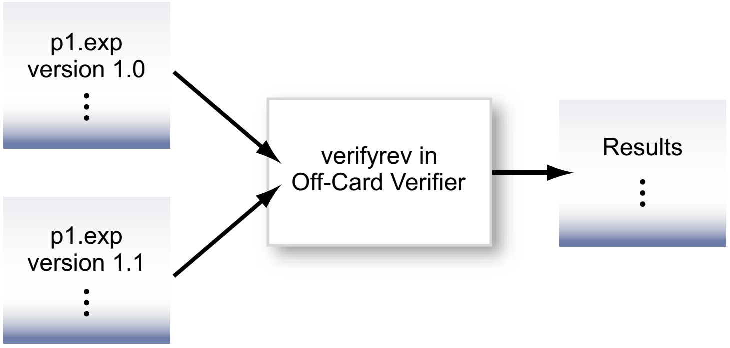
Argument	Description
<export file>	Fully qualified path and name of the export file.

Command Line Options for Off-Card Verifier Tools. describes additional `verifyexp` options.

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 9-3. 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 the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1*, have been followed.

Figure 9-3   Verifying Binary Compatibility Of Export Files



Running verifyrev

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

To run `verifyrev`:

- Enter the following command:

  `verifyrev.bat` [*options*] *export-file export-file*

  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 describes additional command-line options for the off-card verifier tools.

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 exceptions are the `-package`, `-outfile`, `-digest`, and `-target` options that are available for `verifycap` only.

These options exhibit the same behavior regardless of the tool that calls them.

Table 9-3   `verifycap`, `verifyexp`, `verifyrev` Command Line Options

Option	Description
`-help`	Prints help message.
`-nobanner`	Suppresses banner message.
`-nowarn`	Suppresses warning messages.
`-package` <*package name*>	(*Available for* `verifycap` *only*) Sets the name of the package to be verified.
`-outfile`	(*Available for* `verifycap` *only*) Specifies the name of the output file to store the digest (default: no output file created).

Table 9-3 (Cont.) `verifycap`, `verifyexp`, `verifyrev` Command Line Options

Option	Description
`-digest`	(*Available for* `verifycap` *only*) Specifies the digest to use (default: SHA-256)
`-target`	(*Available for* `verifycap` *only*) Specifies the target platform (3.0.4, 3.0.5 or 3.1.0). When a target is specified, the `verifier` automatically uses an internal copy of the export files corresponding to the specified version and ignores the export files for platform packages provided on the command line. This ensures that the correct version of export files is used and allows the `verifier` to detect some binary incompatibility issues when extending some of the platform classes or interfaces on versions 3.0.4 and 3.0.5 of the platform. Note that using this option to specify the target still requires that you provide the export files for all other packages used by the CAP file. If no target is specified, the export files for all the packages used by the CAP file must be provided on the command line.
`-verbose`	Enables verbose mode.
`-version`	Prints version number and exit.
`-C` *command-options-file* or `--commandoptionsfile` *command-options-file*	Optional. Specifies a file containing command-line options.

10

Using Cryptography Extensions

This chapter describes how to use the basic security and cryptography classes.

> **Note:**
>
> Some security and cryptography classes may not be available in all source bundles.

This chapter contains the following sections:

- Overview of Using Cryptography Extensions
- Supported Cryptography Classes
- Instantiating the Classes

Overview of Using Cryptography Extensions

A selection of Security and Cryptography classes are supported by the simulator (`cref`). The support for security and cryptography enables you to:

- Generate message digests using the `SHA1` and `SHA256` algorithms.
- 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`, `HMAC`, `ECC`, and `RSA` algorithms
- Encrypt and decrypt data with the keys using the `AES`, `DES`, `3DES`, and `RSA` algorithms.
- Generate and verify signatures using `MAC`, `CMAC`, `HMAC,DSA`, `ECDSA`, and `RSA` algorithms.
- Generate sequences of random bytes
- Generate checksums
- Use part of a message as padding in a signature block
- Generate derived data using `KDF` in Counter mode and PRF for `TLSv1.2` algorithms

> **Note:**
>
> `DES` is also known as single-key `DES`. `3DES` is also known as triple-`DES`.

Refer to the following publications, for more information on the cryptographic algorithms and schemes:

- 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 (Optimal Asymmetric Encryption Padding) Encryption Scheme*": http://www.emc.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
- For SHA256 — *"Secure Hash Standard"*, FIPS Publication 180-2: http://www.itl.nist.gov
- For HMAC — *"Keyed-Hashing for Message Authentication"*, RFC-2104
- For KDF in Counter mode — *"Key Derivation Function in Counter Mode"*, NIST SP 800-108
- For PRF of TLS—*"Pseudo Random Function"*, TLS version 1.2 defined in IETF RFC 5246
- For DSA — *"Digital Signature Algorithm"*, Standard, NIST FIPS 186.

Supported Cryptography Classes

The implementation of security and cryptography in the simulator supports the use of the following classes:

- javacardx.crypto.AEADCipher
- javacardx.crypto.Cipher
- javacard.security.Checksum
- javacardx.security.derivation.DerivationFunction
- javacardx.security.cert.CertificateParser
- javacard.security.InitializedMessageDigest
- javacard.security.KeyAgreement
- javacard.security.KeyBuilder
- javacard.security.KeyPair
- javacard.security.MessageDigest
- javacard.security.RandomData
- javacard.security.Signature
- javacard.security.SignatureMessageRecovery

Table 10-1 lists the cryptography algorithms that are implemented for the simulator.

Table 10-1 Algorithms Implemented by the Cryptography Classes

Class	Algorithm
AEADCipher	Supports ALG_AES_CCM and ALG_AES_GCM (supports only the 12 byte IV length, which is the value recommended by NIST)
Checksum	• 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	• 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. • ALG_AES_XTS—provides a cipher using AES in XEX Tweakable Block Cipher with Ciphertext Stealing (XTS) mode as defined in IEEE Std 1619. Only the variant with two AES keys of 128-bit length is supported. • ALG_AES_CFB—provides a cipher using AES in Cipher Feedback (CFB) mode. • AEADCipher—Supports ALG_AES_CCM and ALG_AES_GCM (supports only the 12 byte IV length, which is the value recommended by NIST)
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. Provides for SHA1 and SHA256.
KeyAgreement	• ALG_EC_SVDP_DH—elliptic curve secret value derivation primitive, Diffie-Hellman version, per [IEEE P1363]. • ALG_EC_SVDP_DHC—elliptic curve secret value derivation primitive, Diffie-Hellman version, with cofactor multiplication, per [IEEE P1363].
KeyBuilder	The algorithms define the key lengths for: • 128-bit AES • 64-bit DES • 112-, 128-, 160-, 192-bit ECC • 128-bit DES3 • 512-bit RSA • Up to 512-bit HMAC
KeyPair	The algorithms define the key lengths for: • 112-, 128-, 160-, 192-bit ECC • 512-bit RSA
MessageDigest	Message digest algorithm SHA1 and SHA256

Table 10-1 (Cont.) 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	• ALG_DES_MAC8_ISO9797_M2—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. • ALG_RSA_SHA_PKCS1—encrypts the 20 byte SHA1 digest using RSA. The digest is padded according to the PKCS#1 (v1.5) scheme. • ALG_AES_MAC_128_NOPAD—generates a 16-byte MAC using AES with blocksize 128 in CBC mode and does not pad input data. • ALG_ECDSA_SHA—signs/verifies the 20-byte SHA digest using ECDSA. • ALG_AES_CMAC_128 • ALG_HMAC_SHA1 and ALG_HMAC_SHA_256 — generates an HMAC using the steps found in RFC 2104 using the SHA1 and SHA-256 standards, respectively as the hashing algorithm.
SignatureMessageRecovery	• ALG_RSA_SHA_ISO9796_MR—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.
DerivationFunction	• ALG_KDF_COUNTER_MODE—implements KDF in Counter Mode defined in NIST SP 800-108 (recommendation for Key Derivation Using Pseudorandom Functions) with HMAC-SHA-1 or HMAC-SHA-256 as PRF and with HMAC key up to 512 bits size. • ALG_PRF_TLS12—implments the TLS version 1.2 Pseudo Random Function defined in IETF RFC 5246) with HMAC Key up to 512 bits size
CertificateParser	• TYPE_X509_DER—parser for X.509 v1, v2, and v3 DER-encoded certificates (see RFC 5280). • ALG_RSA_SHA_PKCS1—is the supported signature algorithm.

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 10-1 or that is not implemented in this release, getInstance throws a CryptoException with reason code NO_SUCH_ALGORITHM.

Part II

Programming With the Development Kit

This part of the user guide provides solutions for various programming issues. It contains the following chapters:

11 Using Object, CAP File, and Applet Deletion

This chapter describes how to use the object deletion mechanism and the CAP file, and applet deletion features of the Java Card Platform, Version 3.1.
This chapter includes the following topics:

- Object Deletion Mechanism
- CAP File and Applet Deletion

Object Deletion Mechanism

The object deletion mechanism on the Java Card Platform, Version 3.1 reclaims memory that is being used by "unreachable" objects. Objects become unreachable for a number of reasons such as static or instance fields having missing pointers, missing variable references (not only fields), or when the object is orphaned in an island of isolation. An applet object is reachable until it is successfully deleted.

The object deletion mechanism is not like garbage collection in standard Java technology applications due to space and time constraints. The amount of available RAM on the card is limited. In addition, because the object deletion mechanism is applied to objects stored in persistent memory, it must be used sparingly. EEPROM writes are very time-consuming operations and only a limited number of writes can be performed on a card.

Due to these limitations, the object deletion mechanism in Java Card technology is not automatic: it is performed only when an applet requests it. Use the object deletion mechanism sparingly and only when other Java Card technology-based facilities are cumbersome or inadequate.

Requesting the Object Deletion Mechanism

Although any applet on the card can request it, only the Java Card Runtime Environment (Java Card RE) can start the object deletion mechanism. The applet requests the object deletion mechanism with a call to the `JCSystem.requestObjectDeletion()` method.

In the following code example, the method updates the buffer capacity to the given value. If it is not empty, the method creates a new buffer and removes the old one by requesting the object deletion mechanism.

```
/**
* The following method updates the buffer size by removing
* the old buffer object from the memory by requesting
* object deletion and creates a new one with the
* required size.
*/
```

```
void updateBuffer(byte requiredSize){
    try{
        if(buffer != null && buffer.length == requiredSize){
            //we already have a buffer of required size
            return;
        }
        JCSystem.beginTransaction();
        byte[] oldBuffer = buffer;
        buffer = new byte[requiredSize];
        if (oldBuffer != null)
            JCSystem.requestObjectDeletion();
        JCSystem.commitTransaction();
    }catch(Exception e){
        JCSystem.abortTransaction();
    }
}
```

Object Deletion Mechanism Usage Guidelines

The following guidelines describe possible scenarios when the object deletion mechanism might or might not be used:

- When throwing exceptions, avoid creating new exception objects and relying on the object deletion mechanism to perform cleanup. Try to use existing exception objects.
- Try not to create objects in method or block scope. This is acceptable in standard Java technology applications, but is an incorrect use of the object deletion mechanism in Java Card technology-based applications.
- Use the object deletion mechanism when a large object, such as a certificate or key, must be replaced with a new one. In this case, instead of updating the old object in a transaction, create a new object and update its pointer within the transaction. Then, use the object deletion mechanism to remove the old object.
- Use the object deletion mechanism when object resizing is required, as shown in the example in Requesting the Object Deletion Mechanism.

CAP File and Applet Deletion

In the Java Card Platform, Version 3.1, the installer deletes CAP files and applets from the card's memory. Once the installer is selected, it can receive requests from the terminal, in the form of an APDU, to delete CAP files and applets. Requests to delete an applet or CAP file cannot be sent from an applet on the card.

The following sections describe programming guidelines that will help you create CAP files and applets that are more easily removed:

- Developing Removable CAP files
- Writing Removable Applets

Developing Removable CAP File

When a CAP file is deleted, all of its code is removed from the card's memory. A CAP file is eligible for deletion only if there are no dependencies on it, including:

- CAP files that are dependent on the CAP file to be deleted
- Applet instances or objects that either belong to the CAP file, or that belong to a CAP file that depends on the CAP file to be deleted

CAP file deletion will not succeed if any of the following conditions exist:

- A reachable instance of a class belonging to the CAP file exists on the card.
- Another CAP file on the card depends on the CAP file.
- A reset or power failure occurs after the deletion process begins, but before it completes.

To ensure that a CAP file can be easily removed from the card, avoid writing and downloading other CAP files that might be dependent on it. If other CAP files on the card depend on it, you must remove all dependent CAP files before you can remove this CAP file from the card memory.

Writing Removable Applets

Deleting an applet means that the applet and all of its child objects are deleted. Applet deletion fails if any of the following conditions exist:

- Any object owned by the applet instance is referenced by an object owned by another applet instance on the card.
- Any object owned by the applet instance is referenced from a static field in any package on the card.
- The applet is active on the card.

If you are writing an applet that is deemed to be short lived and is to be removed from the card after performing some operation, follow these guidelines to ensure that the applet can be easily removed:

- Write cooperating applets if shareable objects are required. To reduce coupling between applets, try to obtain shareable objects on a per-use basis.
- If interdependent applets are required, make sure that these applets can be deleted simultaneously.
- Follow one of the following guidelines when static reference type fields exist:
  - Ensure there is a mechanism available in the applet to disassociate itself from these fields before applet deletion is attempted.
  - Ensure that the applet instance and code can be removed from the card simultaneously (that is, by using applet and package deletion).

The AppletEvent.uninstall Method

When an applet needs to perform some important actions prior to deletion, it might implement the `uninstall` method of the `AppletEvent` interface. An applet might find it useful to implement this method for the following types of functions:

- Release resources such as shared keys and static objects
- Backup data into another applet's space
- Notify other dependent applets

Calling uninstall does not guarantee that the applet will be deleted. The applet might not be deleted after the completion of the uninstall method in some of these cases:

- Other applets or packages are still dependent on this applet.
- Another applet that needs to be deleted simultaneously cannot currently be deleted.
- The package that needs to be deleted simultaneously cannot currently be deleted.
- A tear occurs before the deletion elements are processed.

To ensure that the applets are deleted, implement the uninstall method defensively. Write your applet with these guidelines in mind:

- The applet continues to function consistently and securely if deletion fails.
- The applet can withstand a possible tear during the execution.
- The uninstall method can be called again if deletion is reattempted.

The following example shows such an implementation:

```
public class TestApp1 extends Applet implements AppletEvent{
    // field set to true after uninstall
    private boolean disableApp = false;
    ...
    public void uninstall(){
        if (!disableApp){
            JCSystem.beginTransaction();  //to protect against tear
            disableApp = true;            //mark as uninstalled
            TestApp2SIO.removeDependency();
            JCSystem.commitTransaction();
        }
    }
    public boolean select(boolean appInstAlreadyActive) {
        // refuse selection if in uninstalled state
        if (disableApp) return false;
        return true;
    }
    ...
}
```

12
Working with Logical Channels

The Java Card Platform, Version 3.1 can support up to twenty logical channels per active interface. Logical channels allow the concurrent execution of multiple applications on the card, allowing a terminal to handle different tasks at the same time.
This chapter includes the following topics:

Dual Interface Cards

On dual interface cards, each interface can handle up to twenty independent logical channels. Channel management commands only affect the logical channels in the interface where the commands are issued.

See the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.1* for more information on logical channels, their implementation, and logical channel terminology.

Applets and Logical Channels

If you design your applets to take advantage of multi-session functionality, they can interoperate from different channels and can be selected multiple times in different channels. For example, the card might handle security information on one channel, while data is accessed on a second channel, while the third channel takes care of data encoding operations.

By following this design, it is possible to access information owned by a different applet without having to deselect the currently selected applet that is handling session information. Thus, you avoid losing your session-specific security data, which is usually stored in `CLEAR_ON_DESELECT` RAM memory.

Non-MultiSelectable Applets

An error is returned to the terminal when an applet that is not designed to be aware of multiple channels is either selected more than once on different channels or is selected concurrently with other applets in the same package.

You can have several non-multiselectable applets operating simultaneously on different channels, as long as they do not interfere with each other's data while they are active. For example, you can open up to 4 channels and run a distinct applet on each as long as they do not interoperate. You can control their operation by multiplexing commands into the APDU communications channel. If the applets are independent of each other, then the results will be the same as if each of these applets were running one at a time, each in a separate session.

The MultiSelectable Interface

For an applet to be selectable on multiple channels at the same time, or to have another applet belonging to the same package selected simultaneously, it must implement the `javacard.framework.MultiSelectable` interface. Implementing this interface allows the applet to be informed when it has been selected more than once or when applets in the same package are already selected during applet activation.

> **Note:**
>
> If an applet in any package implements the `MultiSelectable` interface, then all applets in the package must also implement the `MultiSelectable` interface. It is not possible to have multiselectable and non-multiselectable applets in the same package.

The `MultiSelectable` interface contains a `select` and a `deselect` method to manage multiselectable applets. These methods are described in the following topics:

- Selection for MultiSelectable Applets
- Deselection for MultiSelectable Applets

Selection for MultiSelectable Applets

The `MultiSelectable` interface defines one method to be invoked instead of `Applet.select()` when the applet being selected, or any other applet in its package, is already selected on another logical channel:

```
public boolean MultiSelectable.select(boolean appInstAlreadySelected)
```

The `MultiSelectable.select(boolean)` method informs the applet instance if it is selected more than once on different channels, or if another applet in the same package is selected on another channel on any interface. The parameter `appInstAlreadySelected` is `true` if the applet is selected on a different channel. It is `false` if it is not selected. The method can return either `true` or `false` to accept or reject applet selection.

This method can be called as a result of issuing a `SELECT FILE` or a `MANAGE CHANNEL OPEN` APDU command used to select an applet. If the applet is not selected, then the `appInstAlreadySelected` parameter is passed as `false` to signal an applet activation event. If the applet is subsequently selected on another channel, `MultiSelectable.select(boolean)` is called again, but this time, the `appInstAlreadySelected` parameter is passed as `true`, to indicate that the applet is already active.

Deselection for MultiSelectable Applets

The MultiSelectable interface defines one method to be invoked instead of Applet.deselect() when the applet being deselected, or any other applet in its package, is already selected on another logical channel:

```
public void MultiSelectable.deselect(boolean appInstStillSelected)
```

The MultiSelectable.deselect(boolean) method informs the applet instance if it is being deselected on the logical channel while the same applet instance or another applet in the same package is still active on another channel on any interface. The parameter appInstStillSelected is true if the applet remains active on a different channel. It is false if it is not active on another channel, indicating that this is the last remaining active instance of the applet.

This method can be called as the result of a MANAGE CHANNEL CLOSE or a SELECT FILE APDU command. If the applet remains active on a different channel, the appInstStillSelected parameter is passed as true.

If the MultiSelectable.deselect(boolean) method is called, it means that either an instance of this applet or another applet from the same package remains active on another channel, so CLEAR_ON_DESELECT transients are not cleared.

Only when the last applet instance from the entire package is deselected does a call to Applet.deselect() occur, resulting in the erasure of CLEAR_ON_DESELECT transients.

Writing Applets for Concurrent Logical Channels

This section describes how to write a multiselectable applet that will perform various tasks based on whether it is selected. The code samples in this section show how to extend the applet to implement the MultiSelectable interface and how to implement the MultiSelectable.select(boolean) and deselect(boolean) methods. The code samples also show how to use the Applet.select() and deselect() methods to work with multiselectable applets.

To take advantage of multiple channel operation, an applet must implement the javacard.framework.MultiSelectable interface. For example:

```
public class SampleApplet extends Applet
    implements MultiSelectable {
    ...
    }
```

The new applet needs to provide implementation for the MultiSelectable.select(boolean) and MultiSelectable.deselect(boolean) methods. These methods are responsible for encoding the behavior that the applet needs during a selection event if either of the following situations occurs:

- The applet is already selected on a different channel.
- One or more applets from the same package are also selected on different channels.

The behavior to be encoded might include initializing applet state, accepting or rejecting the selection request, or clearing data structures in case of deselection:

```
public boolean select(boolean appInstAlreadySelected) {
    // Implement the logic to control applet selection
    // during a multiselection situation
    ...
}
public void deselect(boolean appInstStillSelected) {
    // Implement the logic to control applet deselection
    // during a multiselection situation
    ...
}
```

> **Note:**
>
> The applet is still required to implement the `Applet.select()` and the `Applet.deselect()` methods in addition to the `MultiSelectable` interface. These methods handle applet selection and deselection behavior when a multiselection situation does not happen.

Related Topics

- MultiSelectable Applet Example
- Handling Channel Information on APDU Commands
- Writing ISO 7816-4:2005 Compliant Applets
- Non-MultiSelectable Applets and Shareable Objects
- ISO 7816-4:2005 Specific APDU Commands for Logical Channel Management

MultiSelectable Applet Example

In this example, the multiselectable applet, `SampleApplet`, must initialize the following two arrays of data when it is selected:

- An array of package data to be initialized when the first applet in the package becomes active
- An array of private applet data to be initialized upon applet instance activation

You can make these distinctions in your code because the `MultiSelectable` interface allows the applet to recognize the circumstances under which it is selected.

Also, the applet has the following requirements:

- Clear the package data once no applet in the package is active
- Clear the applet private data when the applet instance is deselected

The following methods are responsible for clearing and setting the data:

```
//dataType parameter as above
final static byte DATA_PRIVATE      = (byte)01;
final static byte DATA_PACKAGE      = (byte)02;
...
public void initData(byte[] dataArray, byte dataType) {
    ...
}
public void clearData(byte[] dataArray) {
    ...
}
```

To achieve the behavior specified above, you must modify the selection and deselection methods in your sample applet.

The code for `Applet.select()`, which is invoked when this applet is the first to become active in the package, can be implemented like this:

```
public boolean select() {

    // First applet to be selected in package, so
    // initialize package data and applet data
    initData(packageData, DATA_PACKAGE);
    initData(privateData, DATA_PRIVATE);
    return true;
}
```

Likewise, the implementation of the method `MultiSelectable.select(boolean)` must determine whether the applet is already active. According to its definition, this method is called when another applet within this package is active. `MultiSelectable.select(boolean)` can be implemented so that if `appInstAlreadySelected` is `false`, the applet private data can be initialized. For example:

```
public boolean select(boolean appInstAlreadySelected) {
    // If boolean parameter is false,
    // then we have applet activation
    // Otherwise, no applet activation occurs.
    if (appInstAlreadySelected == false) {
        // Initialize applet private data, upon activation
        initData(privateData, DATA_PRIVATE);
    }
    return true;
}
```

In the case of deselection, the applet data must be cleared. The method MultiSelectable.deselect(boolean) can be implemented so that it clears applet data only if the applet is no longer active. For example:

```
public void deselect(boolean appInstStillSelected) {

    // If boolean parameter is false, then applet is no longer
    // active.  It is O.K. to clear applet private data.
    if (appInstStillSelected == false) {
        clearData(privateData);
    }
}
```

If this applet is the last one to be deactivated from the package, it also must clear package data. This situation results in a call to Applet.deselect(). This method can be implemented like this:

```
public void deselect() {
    // This call means that the applet is no longer active and
    // that no other applet in the package is.  Data for both
    // applet and package must be cleared.
    clearData(packageData);
    clearData(privateData);
}
```

Handling Channel Information on APDU Commands

APDU commands follow the ISO/IEC 7816-4:2013 specifications to encode logical channel information in the CLA byte. The CLA byte encoding is divided into two spaces:

- Interindustry —Used by all ISO/IEC 7816-4:2013- defined commands
- Proprietary — Used by Java Card technology to encode application- specific commands

The CLA byte encoding is divided into two classes:

- Type 4 commands — Encode legacy ISO/IEC 7816-4 logical channel information
- Type 16 commands — Defined by the ISO/IEC 7816-4:2013 specification to encode information for additional 16 logical channels in the card.

Type 4 logical channels occupy the range of [0...3], while Type 16 logical channels go in the range of [4...19], that is, the value encoded in the CLA byte plus four, as it is used in SELECT, MANAGE CHANNEL and other proprietary or ISO commands.

However, a note of caution: while the MANAGE CHANNEL command CLA byte follows the encoding as described below, its P2 parameter does not. The logical channel numbers in its P2 parameter are correctly encoded in the range of [0...19].

The CLA byte encoding follows the following rules:

- Interindustry Space

- Proprietary Java Card Technology Space
- Logical Channels
- APDU Command Type Identification

Interindustry Space

CLA Remarks

`0x0X` Type 4, last or only command in chain

`0x1X` Type 4, not last command in chain (paired with `0x0X`)

`0x2X` Reserved for Future Use

`0x3X` Reserved for Future Use

`0x4X` Type 16, no SM, last or only command in chain

`0x5X` Type 16, no SM, not last command in chain (paired with `0x4X`)

`0x6X` Type 16, SM, last or only command in chain

`0x7X` Type 16, SM, not last command in chain (paired with `0x06X`)

The encoding details are as follows.

Type 4:

```
b8 b7 b6 b5 b4 b3 b2 b1
0  0  0  x  y  y  z  z
```

Type 16:

```
b8 b7 b6 b5 b4 b3 b2 b1
0  1  y  x  z  z  z  z
```

Notation:

`x` = Command Chaining bit

- `0` = last or only command
- `1` = command chaining

`y` = Secure Messaging indicator, see ISO7816-4:2003 section 6 for further information.

`z` = Logical channel indicator

Type 4 supports logical channels [0..3]

Type 16 supports logical channels [0..15], which are mapped to logical channels [4..19]

Proprietary Java Card Technology Space

CLA Remarks

`0x8X` Type 4, last or only command in chain

`0x9X` Type 4, not last command in chain (paired with 0x8X)

`0xAX` Type 4, last or only command in chain

`0xBX` Type 4, not last command in chain (paired with 0xAX)

`0xCX` Type 16, no SM, last or only command in chain

`0xDX` Type 16, no SM, not last command in chain (paired with 0xCX)

`0xEX` Type 16, SM, last or only command in chain

`0xFX` Type 16, SM, not last command in chain (paired with 0xEX)

The encoding details are as follows.

Type 4:

b8	b7	b6	b5	b4	b3	b2	b1
1	0	N/A	x	y	y	z	z

Type 16:

b8	b7	b6	b5	b4	b3	b2	b1
1	1	y	x	z	z	z	z

Logical Channels

When an APDU command is received, the card processes it and determines whether the command has logical channel information encoding. If logical channel information is encoded, then the card sends the APDU command to the respective channel. All other APDU commands are forwarded to the card's basic channel (`0`).

The `X` nibble is responsible for logical channels and secure message encoding. Only the two least significant bits of the nibble are used for channel encoding, which ranges from `0` to `3`. For example, the command `0x21` forwards the command to the card's basic channel (`0`), because the CLA byte with the nibble `0x2X` does not contain logical channel information.

Just as the deselection and selection mechanisms must be written to take into consideration a multiple-channel environment, it is important to write the `Applet.process()` method so that it handles channel information correctly. Due to the fact that some APDUs can be digitally signed, the APDU command is passed to the applet's `process` method as it is sent by the terminal. That means any logical channel information is not cleared and is passed intact to the applet. The applet must deal with this situation.

APDU Command Type Identification

To identify proprietary and interindustry commands, use the `isISOInterindustryCLA` method. This call returns `true` if the CLA byte encoding corresponds to the interindustry space, or `false` if it corresponds to the proprietary space.

```
...
//Applet's process method
public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();

    // check SELECT APDU command
    if (apdu.isISOInterindustryCLA()) {
        if (Applet.selectingApplet()) {
            return;
        } else {
            ISOException.throwIt (ISO7816.SW_CLA_NOT_SUPPORTED);
        }
    }
    ...
```

Writing ISO/IEC 7816-4:2013 Compliant Applets

If your applets must be compliant with the ISO/IEC 7816-4:2013 specification, then you must track the applet security state on each channel where it is active. Additionally, in the case of multiselectable applets, you must copy the state (including its security configuration) when you perform `MANAGE CHANNEL` commands from a channel other than the basic channel.

For example, applets might need to perform some sort of initialization upon activation, as well as cleanup procedures during deactivation. To do these tasks, a multiselectable applet might need to keep track of the channels on which it is being selected during a card session.

To track this information, you need to know the channel on which the task is being performed. Tracking is done by two methods in the Java Card API:

- `APDU` class: `public static byte getCLAChannel();`

  This method returns the origin channel where the command was issued. In case of `MANAGE CHANNEL` or `SELECT FILE` commands, if this method is called within the `Applet.select()`, `MultiSelectable.select(boolean)`, `Applet.deselect()`, or `MultiSelectable.deselect(boolean)` methods, it returns the APDU command logical channel specified in the CLA byte.

- `JCSystem` class: `public static byte getAssignedChannel();`

  This method returns the channel of the currently selected applet. In case of a `MANAGE CHANNEL` command, if this method is invoked inside the `Applet.select()`, `MultiSelectable.select(boolean)`, `Applet.deselect()`, or `MultiSelectable.deselect(boolean)` methods, it returns the channel where the applet to be selected or deselected is assigned to run.

ISO/IEC 7816-4:2013 Compliant Applet Example

This example demonstrates how to copy the security state from the applet selected in the origin channel into the new channel.

In this example, the state information is stored in the array `appState` inside the applet:

```
StateObj appState[MAX_CHANNELS];    // Holds the security state
                                    // for each logical channel
```

You can use the `APDU.getCLAChannel()` and the `JCSystem.getAssignedChannel()` methods to identify if the applet selection case corresponds to an ISO/IEC 7816-4 case where the security state needs to be copied.

Note:

If such an event occurs, it will also be a multiselection situation, where the applet is also selected on the newly opened channel.

In this example, the code to identify the applet selection case is included in the implementation of the `MultiSelectable.select(boolean)` method:

```
public boolean select(boolean appInstAlreadySelected) {
    ...
    // Obtain logical channels information
    // This call returns the channel where
    // the command was issued
    byte origChannel = APDU.getCLAChannel();
    // This call returns the channel where the applet is being
    // selected
    byte targetChannel = JCSystem.getAssignedChannel();
    if (origChannel == targetChannel) {
        // This is a SELECT FILE command.
        // Do processing here.
        ...
    }
    if (origChannel == 0) {
        // This is a MANAGE CHANNEL command from channel 0.
        // ISO 7816-4 state sharing case does not
        // apply here.
        // Do processing here.
        ...
    } else {
        // Since origChannel != 0, the special
        // ISO 7816-4 case applies.
        // Copy security state from origin channel
        // to assigned logical channel.
        appState[targetChannel] = appState[origChannel];
        // Do further processing here
        ...
    }
    ...
}
```

Refer to the API documentation in the in *JC_HOME_SIMULATOR*`\docs` for more information about the APIs.

Non-MultiSelectable Applets and Shareable Objects

Applets that implement `MultiSelectable` are designed to handle calls to Shareable objects across packages when several applets are active on different logical channels. In contrast, an applet that does not implement `MultiSelectable` assumes that it is uniquely selected and its owned objects will not be modified via Shareable interface objects while it is selected. Only when the non-multiselectable applet is in a deselected state can other applets modify its internal data structures.

When you interact with applets that do not implement `MultiSelectable`:

- It is not possible to select more than one applet simultaneously from a package if any of the applets you want to select does not implement the `MultiSelectable` interface.
- It is not possible to invoke methods of a Shareable object belonging to a non-multiselectable applet when an applet, belonging to the same group context, is active.

ISO/IEC 7816-4:2013 Specific APDU Commands for Logical Channel Management

There are two ISO-specific APDU commands that you can use to work with logical channels in a smart card:

- `SELECT FILE` — This command selects the specified applet on the specified channel number. The channel number can be from `0` to `3` and is specified in the lower two bits of the CLA byte. If the channel is closed, it is opened and the specified applet is selected on the channel. `SELECT FILE` commands are forwarded to the newly selected applet.
- `MANAGE CHANNEL` — This command can be used to open a new channel from another channel or close it. It allows you to specify the channel to be used or to allow the smart card to select the channel. Like `SELECT FILE`, this command uses the lower two bits of the CLA byte to specify the channel number. `MANAGE CHANNEL` commands are not forwarded to the applet.

When you work with these commands, keep the following guidelines in mind:

- Origin logical channel values are encoded in the two least significant bits of the CLA byte.
- Logical channel values have a valid range of [`0`..`19`] only.
- Logical channel `0` is known as the *basic channel*, and it cannot be closed.
- At card reset, the basic channel (channel `0`) is open. All other channels (`1`, `2`, ...`19`) are closed.

The `MANAGE CHANNEL` and `SELECT FILE` commands are read by the Java Card RE dispatcher, which performs the functions specified by the commands, including the following:

- Managing logical channels
- Deselecting applets
- Selecting applets

MANAGE CHANNEL OPEN

In response to the `MANAGE CHANNEL OPEN` command, the dispatcher follows this procedure:

1. If the origin channel is not open, an error is returned.
2. Determines whether the channel is open or closed. If the channel is open, an error is returned.

3. Opens the channel.
4. If the origin channel is `0`, the default applet (if there is one) is selected in the new channel.
5. If the origin channel is not `0`, the selected applet on the origin channel becomes the selected applet in new channel.

This `MANAGE CHANNEL OPEN` command opens a new channel from channel encoded in `Q`:

CLA	INS	P1	P2	Lc	Data	Le	Data	SW1	SW2
`0xQ`	`0x70`	`00`	`00`	`0`	`-`	`1`	`0x0R`	`0x90`	`00`

:

CLA	INS	P1	P2	Lc	Data	Le	SW1	SW2	SW2
`0xQ`	`0x70`	`00`	`0xR`	`0`	`-`	`0`	`0x90`	`00`	`00`

This command produces the following results:

- If channel encoded in `Q` is the basic channel (channel `0`), the card's default applet is selected on channel encoded in `R`. No applet is selected if no default applet is defined.
- If channel encoded in `Q` is other than the basic channel (channels `1`, `2`, ...`19`), the selected applet on channel encoded in `Q` becomes the current applet selected on channel `R`.
- The applet on channel encoded in `R` can either accept or reject selection.

This command returns an error under the following circumstances:

- The applet does not implement the `javacard.framework.MultiSelectable` interface, when an attempt to select the applet in more than one channel takes place.
- The applet rejects selection or throws exception.
- No channel is available.
- Channel encoded in `Q` is not open.

MANAGE CHANNEL CLOSE

In response to the `MANAGE CHANNEL CLOSE` command, the dispatcher follows this procedure:

1. If the origin channel is not open, an error is returned.
2. If the channel to be closed is `0`, an error is returned.
3. If the channel to be closed is not open or not available, a warning is thrown.
4. Deselects the applet in the channel to be closed.
5. Closes the logical channel.

This `MANAGE CHANNEL CLOSE` command closes channel `R` from channel `Q`:

CLA	INS	P1	P2	Lc	Data	Le	SW1	SW2	SW2
0xQ	0x70	0x80	0xR	0	-	0	0x90	00	00

This command closes channel R. Channel R must not be the basic channel (it can be channel 1, 2, ...19 only).

This command returns an error in the following circumstances:

- Channel encoded in R is the basic channel.
- Channel encoded in Q is not open.

It returns a warning if channel R is not open.

SELECT FILE

In response to the SELECT FILE command, the dispatcher follows this procedure:

1. If the specified channel is closed, open the channel.
2. Deselect currently selected applet in channel if needed.
3. Select specified applet in the channel.

This SELECT FILE command selects an applet on channel R:

CLA	INS	P1	P2	Lc	Data	Le	SW1	SW2
0x0R	0xA4	0x04	0x00	(AID len)	(AID)	0	0x90	00

This command produces the following results:

- Channel encoded in R can be any channel (opened or unopened), including the basic channel.
- The applet identified in the Data section becomes the selected applet on channel R.
- If channel encoded in R is not open, this command opens channel R.
- If channel encoded in R is open, this command changes the selected applet in the channel to the one specified.

This command returns an error in the following circumstances:

- The applet cannot be found or is not available. The current applet is left selected and an error is returned.
- An active applet belonging to the same package does not implement the javacard.framework.MultiSelectable interface, or if the applet to be selected does not implement this interface.
- Channel encoded in R is not available.

13
Using Java Card RMI

This chapter describes how to write remote method invocation (RMI) applications and how to use the RMI client-side API for the Java Card Platform, Version 3.1.

Topics:

- Developing RMI Applications for the Java Card Platform
- Programming to the Java Card RMI ClientSide API

Developing RMI Applications for the Java Card Platform

This section describes how to write remote method invocation (RMI) applications for the Java Card Platform, Version 3.1. Because the Java Card specifications state that Java Card RMI is optional, verify that your targeted card supports Java Card RMI before using these APIs.
This section includes the following topics:

- Steps to Develop an RMI Applet for the Java Card Platform
- RMI Program Example
- Add Security Support

Steps to Develop an RMI Applet for the Java Card Platform

There are three main steps to develop an RMI applet:

1. Define remote interfaces.
2. Develop classes implementing the remote interfaces.
3. Develop the `main` class for the applet. For a simple applet, the main class of the applet can also be the class implementing the remote interface.

This section includes the following topics:

- Generating Stubs
- Running a Java Card RMI Applet

Generating Stubs

The Java Card RMI Client framework requires stubs only when the `remote_ref_with_class` format is used for passing remote references. These stubs of remote classes of applets must be pre-generated and available on the client. When the `remote_ref_with_interfaces` format is used, stubs are not necessary.

In this example, the Java RMI Compiler (`rmic`) is used to generate these stubs.

Following is the command to run the `rmic`:

`rmic -v1.2 -classpath` *path* `-d` *output_dir class_name*

In the command:

- *path* includes the path to the remote class of your sample applet and to the file, `JC_HOME_TOOLS\tools.jar`
- *output_dir* is the directory in which to place the resulting stubs
- *class_name* is the name of the remote class
- The `-v1.2` flag is required by the RMI client framework for the Java Card Platform, Version 3.1

The `rmic` must be called for each remote class in your applet.

> **Note:**
>
> You need to generate stubs only for remote classes that list a remote interface in their `implements` clause.

The file `tools.jar`, provided in the Java Card Development Kit contains compiled implementations of packages `javacard.framework`, `javacard.security`, `javacardx.biometry, javacardx.external` and `javacardx.framework.tlv`. Classes in these packages might be referenced by Java Card RMI applets and thus might be needed by the `rmic` to generate stubs.

Running a Java Card RMI Applet

The server part (the Java Card RMI-enabled applet) can be run on the C-language Java Card RE, for which the following standard procedures apply:

- The applet must be installed first by using the installer applet.
- After the applet is installed, the EEPROM state can be saved and used to run the Java Card RE against the Java Card RMI client.

RMI Program Example

The RMI program example is the Java Card platform equivalent of "Hello World." It is a program that manages a counter remotely, and is able to decrement, increment, and return the value of the counter.

This section includes the following topics:

- Main Program
- Sample Applet
- Client Example
- Card Terminal Interaction

Main Program

As for any Java Card RMI program, the first step is to define the interface to be used as contract between the server (the Java Card technology-based application) and its clients (the terminal applications):

```
package examples.purse;
import java.rmi.*;
import javacard.framework.*;
public interface Purse extends Remote {
    public static final short MAX_AMOUNT = 400;
    public static final short REQUEST_FAILED = 0x0102;
    public short debit(short amount) throws RemoteException,
UserException;
    public short credit(short amount) throws RemoteException,
    UserException;
    public short getBalance() throws RemoteException, UserException;
}
```

This is a typical Java Card RMI interface in the following ways:

- The interface type extends the `java.rmi.Remote` interface. This interface is a tagging interface that identifies the interface as defining a remotely accessible object.
- Every method in the interface must be declared as throwing a `RemoteException` or one of its superclasses (`IOException` or `Exception`). This exception is required to encapsulate all the communication problems that might occur during a remote invocation of the method. In addition, the `credit`, `debit`, and `getBalance` methods also throw the `UserException` to indicate application-specific errors.
- The interface can also define values for constants that might be used in communication between the client and the server. The `Purse` interface defines a constant `MAX_AMOUNT` that represents the maximum allowed value for the transaction amount parameter. It also defines a reason code `REQUEST_FAILED` for the `UserException` qualifier.

Related Topics

- Implement a Remote Interface
- Define the Constructor for the Remote Object
- Provide an Implementation for Each Remote Method

Implement a Remote Interface

This code sample provides an implementation for the remote interface. The implementation runs on a Java Card Platform, so it can use only features that are supported by a Java Card Platform, Version 3.1.

```
package examples.purse;

import javacard.framework.*;
import javacard.framework.service.*;
import java.rmi.*;

public class PurseImpl extends CardRemoteObject implements Purse {
    private short balance;

    PurseImpl() {
        super();
        balance = 0;
    }

    public short debit(short amount) throws RemoteException,
UserException {
        if ((amount < 0) || (amount > MAX_AMOUNT))
            UserException.throwIt(REQUEST_FAILED);
        balance -= amount;
        return balance;
    }

    public short credit(short amount) throws RemoteException,
UserException {
        if ((amount < 0) || (balance < amount))
            UserException.throwIt(REQUEST_FAILED);
        balance += amount;
        return balance;
   }

   public short getBalance() throws RemoteException, UserException {
       return balance;
   }
}
```

Here, the remote interface is the `Purse` interface, which declares the remotely accessible methods. By implementing this interface, the class establishes a contract between itself and the compiler, by which the class promises that it will provide method bodies for all the methods declared in the interface:

```
public class PurseImpl extends CardRemoteObject implements Purse
```

The class also extends the `javacard.framework.service.CardRemoteObject` class. This class provides basic support for remote objects, and in particular the ability to export or unexport an object.

Define the Constructor for the Remote Object

The constructor for a remote class provides the same functionality as the constructor of a non-remote class; it initializes the variables of each newly created instance of the class.

In addition, the remote object instance needs to be exported to make it available to accept incoming remote method requests. By extending `CardRemoteObject`, a class guarantees that its instances are exported automatically upon creation on the card.

If a remote object does not extend `CardRemoteObject` (directly or indirectly), you must explicitly export the remote object by calling the `CardRemoteObject.export` method in the constructor of your class (or in any appropriate initialization method). Of course, this class must still implement a remote interface.

To review, the implementation class for a remote object needs to do the following:

- Implement a remote interface
- Export the object so that it can accept incoming remote method calls

Provide an Implementation for Each Remote Method

The implementation class for a remote object contains the code that implements each of the remote methods specified in the remote interface. For example, the following code is the implementation of the method that debits the purse:

```
 public short debit(short amount) throws RemoteException, UserException
     if (( amount < 0 )||( balance < amount )
         UserException.throwIt(REQUEST_FAILED);
     balance -= amount;
     return balance;
}
```

An operation is only allowed if the value of its parameter is compatible with the current state of the purse object. In this particular case, the application only checks that the amounts handled are positive and that the balance of the purse always remains positive.

In Java Card RMI, the arguments to and return values from remote methods are restricted. The main reason for this limitation is that the Java Card Platform, Version 3.1 does not support object serialization. The following are the rules for the Java Card Platform, Version 3.1:

- The arguments to remote methods can be of any supported integral type (such as `boolean`, `byte`, `short` and `int`), or any single-dimensional arrays of these integral types.

> **Note:**
>
> The `int` type is optionally supported on the Java Card Platform, Version 3.1, so applications that use this type might not run on all platforms.

- The return value from a remote method can be any type supported as arguments, as well as any remote interface type. The method can also return `void`.

On the other hand, object passing in Java Card RMI follows the normal RMI rules:

- By default, non-remote objects are passed by copy, which means that all data members of an object are copied, except those marked `static` or `transient`. In the case of the Java Card Platform, Version 3.1, this rule is trivial to apply, because the only objects concerned are arrays of integral types.
- Remote objects are passed by reference. In the case of the Java Card Platform, Version 3.1, remote objects can only be passed as return values. A reference to a remote object is actually a reference to a stub, which is a client-side proxy for the remote objects. Stubs are needed only when the format `remote_ref_with_class` is used for passing remote references. When another format, such as `remote_ref_with_interfaces`, is used, stubs are not necessary. Stubs are described in Generate the Stubs.

Note:

Even though the semantics of the Java Card Platform, Version 3.1 transient arrays are somewhat similar to transient fields in the Java programming language, different rules apply. The Java Card Platform, Version 3.1 contents are copied in Java Card RMI and passed by value when they are returned from a remote method.

A class can define methods not specified in a remote interface, but they can only be invoked on-card within the Java Card VM and cannot be invoked remotely.

Sample Applet

In the Java Card Platform, Version 3.1, all applications must include a class that inherits from `javacard.framework.Applet`, which will provide an interface with the outside world.

This also applies to applications that are based on remote objects, for two main reasons:

- The remote objects must be instantiated and initialized, which can be done in an applet's `install` method.
- The remote objects must communicate with the outside world, which can be done in an applet's `process` method.

For conversion, an applet should be assigned with an AID known on the client side, `0x00;0x01:0x02:0x03:0x04:0x05:0x06:0x07:0x08`, since this AID is used in the client program.

The following is the basic code for such an applet:

```
package examples.purse;

import javacard.framework.*;
import javacard.framework.service.*;
import java.rmi.*;

public class PurseApplet extends Applet {
    private Dispatcher dispatcher;

    private PurseApplet() {
        // Allocates an RMI service and sets for the Java Card platform
        // the initial reference
        RemoteService rmi = new RMIService(new PurseImpl());
        // Allocates a dispatcher for the remote service
        dispatcher = new Dispatcher((short) 1);
        dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND);
    }

    public static void install(byte[] buffer, short offset, byte length) {
        // Allocates and registers the applet
        (new PurseApplet()).register();
    }
    public void process(APDU apdu) {
        dispatcher.process(apdu);
    }
}
```

Related Topics

- Preparing and Registering the Remote Object
- Processing the Incoming Commands

Preparing and Registering the Remote Object

The `PurseApplet` constructor contains the initialization code for the remote object.

First, a `javacard.framework.service.RMIService` object must be allocated. This service is an object that knows how to handle all the incoming `APDU` commands related to the Java Card RMI protocol. The service must be initialized to allow remote methods on an instance of the `PurseImpl` class. A new instance of `PurseImpl` is created, and is specified as the initial reference parameter to the `RMIService` constructor as shown in the following code snippet. The initial reference is the reference that is made public by an applet to all its clients. It is used as a bootstrap for a client session, and is similar to that registered by a Java RMI server to the Java Card RMI registry.

```
RemoteService rmi = new RMIService(new PurseImpl());
```

Then, a dispatcher is created and initialized. A dispatcher is the glue among several services. In this example, the initialization is quite simple, because there is a single service to initialize:

```
dispatcher = new Dispatcher((short)1);
dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND);
```

Finally, the applet must register itself to the Java Card RE to be made selectable. This is done in the `install` method, where the applet constructor is invoked and immediately registered:

```
(new PurseApplet()).register();
```

Processing the Incoming Commands

Processing incoming commands is entirely delegated to the Java Card RMI service, which knows how to handle all the incoming requests. The service also implements a default behavior for the handling of any request that it does not recognize. In Java Card RMI, the following kinds of requests can be handled:

- **Selection request** — The service responds by sending its initial remote reference
- **Method invocation request** — The service responds by performing the actual method invocation and returning the result

To perform these actions, the service needs privileged access to some resources that are owned by the Java Card RE (in particular, privileged access is needed to perform the method invocation). The applet delegates processing to the Java Card RMI service from its process method as follows:

```
dispatcher.process(apdu);
```

Client Example

Client applications run on a terminal supporting a Java Virtual Machine environment such as Java Platform, Standard Edition or Java Platform, Micro Edition (Java ME).

The PurseClient application interacts with the remote stub classes generated by a stub generation tool and the Java Card platform-specific information managed by the Java Card platform client-side framework located in packages com.sun.javacard.clientlib and com.sun.javacard.rmiclientlib.

The client example below uses standard Java RMIC compiler-generated client-side stubs. The client application as well as the Java Card client-side framework rely on the APDU I/O library for managing and communicating with the card reader and the card on which the Java Card applet PurseApplet resides. This makes the client application very portable on Java SE platforms. See the Working with APDU I/O section for more information on the APDU I/O library.

The following example shows a very simple PurseClient application that is the client application of the Java Card technology-based program PurseApplet:

```
import examples.purse.*;
import javacard.framework.UserException;

public class PurseClient extends java.lang.Object {
    public static void main(java.lang.String[] argv) {
        // arg[0] contains the debit amount
        short debitAmount = (short) Integer.parseInt(argv[0]);
        CardAccessor ca = null;
        try {
            // open and powerup the card
            ca = new ApduIOCardAccessor();
            // create an RMI connector instance for the Java Card platform
            JCRMIConnect jcRMI = new JCRMIConnect(ca);
            byte[] appAID = new byte[]
{0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08};
            // select the Java Card applet
            jcRMI.selectApplet( RMI_DEMO_AID,
JCRMIConnect.REF_WITH_CLASS_NAME );
            or
            jcRMI.selectApplet( RMI_DEMO_AID,
JCRMIConnect.REF_WITH_INTERFACE_NAMES );
            // obtain the initial reference to the Purse interface
            Purse myPurse = (Purse) jcRMI.getInitialReference();
            // debit the requested amount
            try {
                short balance = myPurse.debit ( debitAmount );
            }catch ( UserException jce ) {
                short reasonCode = jce.getReason();


                // process UserException reason information
             }
             // display the balance to user
         }catch (Exception e) {
             e.printStackTrace();
         }finally {
             try {
                 if(ca!=null){
                     ca.closeCard();
                 }
             }catch (Exception e) {
                 e.printStackTrace();
             }
         }
     }
}
```

Related Topics

- Initializing and Shutting Down the Card Connection
- Creating and Using a CardAccessor Object
- Selecting the Java Card Applet and Obtaining the Initial Reference
- Using Remote Objects in Remote Method Invocations
- Generate the Stubs

Initializing and Shutting Down the Card Connection

The client application must open the connection to the card and close it at the end.

Note:

`ApduIOCardAccessor` takes its settings from the file `jcclient.properties`. For example. when the RMIPurse sample demo client application runs, the `JC_HOME_SIMULATOR/samples/classic_applets/RMIPurse/client` directory containing the properties file is included in the `CLASSPATH`. The directory in which you installed the developer's kit is indicated as *`JC_HOME_SIMULATOR`*.

On Microsoft Windows platforms, use backslashes in directory paths, instead of forward slashes.

The following code shows opening and closing the connection using the RMI client framework:

```
      CardAccessor ca = null;
      // The following line initializes card connection according to
      // parameters listed in the jcclient.properties file:
      ca = new ApduIOCardAccessor();
      ...
      // The following line powers down the card and closes the
connection:
      ca.closeCard();
```

Creating and Using a CardAccessor Object

To access the Java Card applet using remote methods, the client application must obtain an instance of the CardAccessor interface. The ApduIO class implements the CardAccessor interface and is included in the framework.

The CardAccessor interface is a platform-independent and framework-independent interface used by the RMI framework for the Java Card platform to communicate with the card. The CardAccessor object is then provided as a parameter during construction of the JavaCardRMIConnect class to initiate an RMI dialog for the Java Card platform as shown in the following code:

```
// create an RMI connection object for the Java Card platform
JavaCardRMIConnect jcRMI = new JavaCardRMIConnect( myCS );
```

Selecting the Java Card Applet and Obtaining the Initial Reference

To invoke methods on the remote objects of PurseApplet on the card, it must first be selected by using the AID as shown in the following code:

```
// select the Java Card applet
byte[] appAID = new byte[] {0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08};
jcRMI.selectApplet( appAID );
```

Then, the client must obtain the initial reference remote object for PurseApplet. JavaCardRMIConnect returns an instance of a stub class corresponding to the PurseImpl class on the card, which implements the Purse interface. The client application knows beforehand that the PurseApplet's initial remote reference implements the Purse interface and therefore casts it appropriately as shown in the following code:

```
// obtain the initial reference to the Purse interface
Purse myPurse = (Purse) jcRMI.getInitialReference();
```

Using Remote Objects in Remote Method Invocations

The client can now invoke remote methods on the initial reference object. The remote methods are declared in the Purse interface. The following code shows the client invoking the debit method.

> **Note:**
>
> A UserException exception thrown by the remote method is caught by the client code in normal Java programming language style.

```
// debit the requested amount
try {
    short balance = myPurse.debit ( debitAmount );
}catch ( UserException jce ) {
```

```
            short reasonCode = jce.getReason();
            // process on card exception reason information
        }
```

Generate the Stubs

The client-side scenario uses rmic generated stubs for the remote classes. For the client application PurseClient to execute correctly on the terminal, it needs these remote stub classes and the remote interface class files it uses to be accessible in its classpath.

The stub class PurseImpl_Stub.class for the PurseImpl class is produced by running the standard JDK compiler. The directory where you installed the developer's kit is indicated by JC_HOME_SIMULATOR. For example, from the examples/purse directory, enter the following command:

```
rmic -classpath ../..;%JC_HOME_TOOLS%/lib/tools.jar -d ../..
-v1.2 examples.purse.PurseImpl
```

This produces a stub class called examples.purse.PurseImpl_Stub.

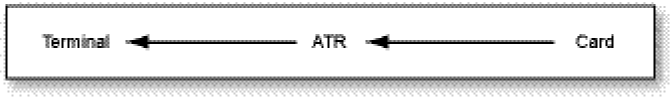
For PurseClient to run correctly on the terminal, the following files must be present in the examples/purse directory and accessible either from its classpath or from class loaders:

- PurseImpl_Stub.class
- Purse.class

Card Terminal Interaction

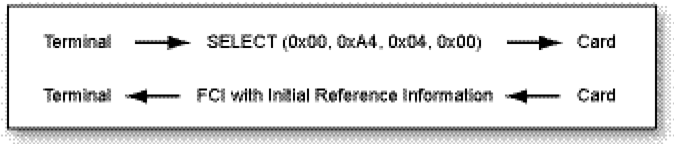
When a Java Card technology-enabled smart card is powered up, the card sends an ATR (Answer to Reset) to the terminal. The Card Accessor returns the value of the ATR to the client program (shown in Figure 13-1).

Figure 13-1 Smart Card Sends an ATR to the Terminal



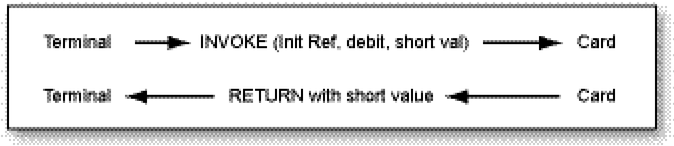
When the PurseClient application calls the selectApplet method of JavaCardRMIConnect, it sends a SELECT APDU command to the card via the CardAccessor object. This results in a File Control Information (FCI) APDU response from the RMIService instance of PurseApplet on the card in a TLV (Tag Length Value) format that includes the initial reference remote object information (shown in Figure 13-2).

Figure 13-2   Terminal Sends a SELECT Command to the Smart Card, Which Returns FCI


Terminal
SELECT (0x00, 0xA4, 0x04, 0x00)
Card
Terminal
FCI with Initial Reference Information
Card

Later, when the `PurseClient` application calls the `debit` method of the remote interface `Purse`, the `PurseImpl_Stub` object sends an `INVOKE` command to the card via the `CardAccessor` object, identifying the remote object reference, interface, method, and parameter data for method invocation. The `RMIService` instance of `PurseApplet` unmarshalls this information and invokes the `debit` method of the `PurseImpl` instance, and returns the return value in the response `RETURN` APDU (shown in Figure 13-3).

Figure 13-3   Terminal Sends an INVOKE Command to the Smart Card, Which Returns a Value



Add Security Support

The previous Sample Applet example is extremely simple and is not realistic. In particular, it does not include any form of security. Users are not authenticated and no transport security is provided. Of course, every smart card that implements the Java Card platform includes such security mechanisms, because they are central to Java Card technology.

The following section describes how to add security support to the `Purse` example.

The `Purse` interface in the package `examples.securepurse` is similar to the `Purse` interface used in the Sample Applet example. In addition, it might include reason codes for exceptions to report security violations to the terminal. This example replaces the `Purse` interface used in theSample Applet example with the following `examples.securepurse` code. The`Purse` interface in the `examples.securepurse` does not include an implementation, which means that, in particular, it does not include any support for security.

The applet keeps its original organization but it also includes additional code that is dedicated to the management of security.

```
package examples.securepurse;

import javacard.framework.*;
import javacard.framework.service.*;
import java.rmi.*;
```

```
public class SecurePurseImpl implements Purse {
    private short balance;
    private SecurityService security;

    SecurePurseImpl(SecurityService security) {
        this.security = security;
    }

    public short debit(short amount) throws RemoteException,
UserException {
        if ((!security
                .isCommandSecure(SecurityService.PROPERTY_INPUT_INTEGRITY)
)
                || (!security
                         .isAuthenticated(SecurityService.PRINCIPAL_CARDHO
LDER)))
           UserException.throwIt(REQUEST_FAILED);
        if (( amount < 0 )|| ( balance < amount ))
            UserException.throwIt(REQUEST_FAILED);
        balance -= amount;
        return balance;
    }

    public short credit(short amount) throws RemoteException,
UserException {
        if ((!security
                .isCommandSecure(SecurityService.PROPERTY_INPUT_INTEGRITY)
)
                || (!security
                        .isAuthenticated(SecurityService.PRINCIPAL_APP_PRO
VIDER)))
           UserException.throwIt(REQUEST_FAILED);
        if (( amount < 0 )||( amount > MAX_AMOUNT ))
           UserException.throwIt(REQUEST_FAILED);
        balance += amount;
        return balance;
    }

    public short getBalance() throws RemoteException, UserException {
        if ((!
security.isAuthenticated(SecurityService.PRINCIPAL_CARDHOLDER))
                && (!security
                        .isAuthenticated(SecurityService.PRINCIPAL_APP_PRO
VIDER)))
            UserException.throwIt(REQUEST_FAILED);
        return balance;
    }
}
```

Related Topics

- Initialize a Security Service
- Use the Service to Check the Current Security Status
- Security Service Example

- More Secure Applet
- Client Changes to Support Security
- CustomCardAccessor Class for Authentication and Signing

Initialize a Security Service

In this example, basic security services (principal identification and authentication, secure communication channel) are provided by an object that implements the `SecurityService` interface. Because a generic remote object must not be dependent on a particular kind of security service, it must take a reference to this object as a parameter to its constructor. This is exactly what happens here, where the reference to the object is stored in a dedicated private field:

```
private SecurityService security ;
```

The `SecurityService` interface is part of the extended application development framework and offers an API that can then be used to check on the current security status.

Use the Service to Check the Current Security Status

In the example, the following are required security behaviors for the applet:

- The `debit` method is authorized only if it is sent through a secure channel that ensures at least the integrity of input data, and if the cardholder is successfully authenticated.
- The `credit` method is authorized only if it is sent through a secure channel that ensures at least the integrity of input data, and if the application issuer is successfully authenticated.
- The `getBalance` method is authorized only if the cardholder or the application issuer is successfully authenticated.

The `SecurityService` provides methods and constants that allow the implementation to perform such checks. For instance, following is the code for the checks on the `debit` method:

```
        if ((!security
                .isCommandSecure(SecurityService.PROPERTY_INPUT_INTEGRITY)
)
                || (!security
                       .isAuthenticated(SecurityService.ID_CARDHOLDER)))
           UserException.throwIt(REQUEST_FAILED);
```

If one of the two conditions is not satisfied, the remote object throws an exception. This exception is caught by the dispatcher and forwarded to the client.

Security Service Example

The following example demonstrates how to implement a security service.

```
package com.sun.javacard.samples.SecureRMIDemo;
```

```
import javacard.framework.*;
import javacard.framework.service.*;

public class MySecurityService extends BasicService implements 
SecurityService {
    // list IDs of known parties...
    private static final byte[] PRINCIPAL_APP_PROVIDER_ID = {0x12, 0x34};
    private static final byte[] PRINCIPAL_CARDHOLDER_ID = {0x43, 0x21};
    private OwnerPIN provider_pin, cardholder_pin = null;
    // and the security-related session flags
    ...
    public MySecurityService() {
        // initialize the PINs
        ...
    }
    public boolean processDataIn(APDU apdu) {
        if(selectingApplet()) {
            // reset all flags
            ...
        }
        else {
            return preprocessCommandAPDU(apdu);
        }
    }
    public boolean isCommandSecure(byte properties) throws 
ServiceException {
        // return the value of appropriate flag
        ....
    }
    public boolean isAuthenticated(short principal) throws 
ServiceException {
        // return the value of appropriate flag
        ....
    }
    private byte authenticated;
    private boolean preprocessCommandAPDU(APDU apdu) {
        receiveInData(apdu);
        if(checkAndRemoveChecksum(apdu)) {


            // set DATA_INTEGRITY flag
        }
        else {
            // reset DATA_INTEGRITY flag
        }
        return false;   // other services may also preprocess the data
     }
     private boolean checkAndRemoveChecksum(APDU apdu) {
         // remove the checksum
         // return true if checksum OK, false otherwise
     }
     public boolean processCommand(APDU apdu) {
         if(isAuthenticate(apdu)) {
             receiveInData(apdu);
             // check PIN
             // set AUTHENTICATED flags
             return true;     //  processing of the command is finished
         }
         else {
             return false;  // this command was addressed to another
             // service - no processing is done
         }
     }
     public boolean processDataOut(APDU apdu) {
         // add checksum to outgoing data
         return false;  // other services may also postprocess outgoing
data
     }
     private boolean isAuthenticate(APDU command) {
         // check values of CLA and INS bytes
     }
}
```

More Secure Applet

The supporting applet also must undergo some significant changes, in particular regarding the initialization of the remote object:

```
package examples.securepurse;

import javacard.framework.*;
import javacard.framework.service.*;
import java.rmi.*;
import com.sun.javacard.samples.SecureRMIDemo.MySecurityService;

public class SecurePurseApplet extends Applet {
    Dispatcher dispatcher;

    private SecurePurseApplet() {
        SecurityService sec;
        // First get a security service
        sec = new MySecurityService();
        // Allocates an RMI service for the Java Card platform and
        // sets the initial reference
        RemoteService rmi = new RMIService(new SecurePurseImpl(sec));
        // Allocates and initializes a dispatcher for the remote object
        dispatcher = new Dispatcher((short) 2);
        dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND);
        dispatcher.addService(sec, Dispatcher.PROCESS_INPUT_DATA);
    }

    public static void install(byte[] buffer, short offset, byte length) {
        // Allocates and registers the applet
        (new SecurePurseApplet()).register();
    }

    public void process(APDU apdu) {
        dispatcher.process(apdu);
    }
}
```

The security service that is used by the remote object must be initialized at some point. Here, this is done in the constructor for the SecurePurseApplet:

```
sec = new MySecurityService();
```

The initialization then goes on with the initialization of the Java Card RMI service. The only new thing here is that the remote object being allocated and set as the initial reference is now a SecurePurseImpl:

```
RemoteService rmi = new RMIService( new SecurePurseImpl(sec) );
```

Next, the dispatcher must be initialized. Here, it must dispatch simple Java Card RMI requests and security-related requests (such as EXTERNAL AUTHENTICATE). In fact, the security service handles these requests directly. First, allocate a dispatcher and inform it that it will delegate commands to two different services:

```
dispatcher = new Dispatcher((short)2);
```

Then, register services with the dispatcher. The security service is registered as a service that performs preprocessing operations on incoming commands, and the Java Card RMI service is registered as a service that processes the command requested:

```
dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND);
dispatcher.addService(sec, Dispatcher.PROCESS_INPUT_DATA);
```

The rest of the class (install and process methods) remain unchanged.

Client Changes to Support Security

The driver client application itself only changes minimally to account for the authentication and integrity needs of SecurePurseApplet. It must also interact with the user for identification. Hence, a subclass of ApduIO_Card_Accessor must be developed to provide these additional interactions and the transport filtering required.

The following code is the new SecurePurseClient application:

```
import examples.purse.*;
import javacard.framework.UserException;

public class PurseClient extends java.lang.Object {
    public static void main(java.lang.String[] argv) {
        // arg[0] contains the debit amount
        short debitAmount = (short) Integer.parseInt(argv[0]);
        CustomCardAccessor cca = null;
        try {
            // open and powerup the card - using CustomCardAccessor
            cca = new CustomCardAccessor();
            // create an RMI connector instance for the Java Card platform
            JCRMIConnect jcRMI = new JCRMIConnect(cca);
            byte[] appAID = new byte[]
{0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08};
            // select the Java Card applet
            jcRMI.selectApplet( RMI_DEMO_AID,
JCRMIConnect.REF_WITH_CLASS_NAME );
```

```
            or
            jcRMI.selectApplet( RMI_DEMO_AID,
JCRMIConnect.REF_WITH_INTERFACE_NAMES );


            // give your PIN
            if (! cca.authenticateUser( PRINCIPAL_CARDHOLDER_ID )){
                throw new RemoteException(msg.getString("msg04"));
            }
            // obtain the initial reference to the Purse interface
            Purse myPurse = (Purse) jcRMI.getInitialReference();
            // debit the requested amount
            try {
                short balance = myPurse.debit ( debitAmount );
            }catch ( UserException jce ) {
                short reasonCode = jce.getReason();
                // process UserException reason information
            }
            // display the balance to user
        }catch (Exception e) {
            e.printStackTrace();
        }finally {
            try {
                if(cca!=null){
                    cca.closeCard();
                }
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Note that the `CustomCardAccessor` instance is now obtained instead of `ApduIOCardAccessor`:

```
         cca = new CustomCardAccessor(new ApduIOCardAccessor());
```

An extra step to authenticate with the `SecurePurseApplet` after `selectApplet` is added. This invokes a new method in `CustomCardAccessor` to interact with the card using the user's credentials:

```
            if (! cca.authenticateUser( PRINCIPAL_CARDHOLDER_ID )) {
                // handle error
            }
```

The rest of `SecurePurseClient` is the same as `PurseClient`.

CustomCardAccessor Class for Authentication and Signing

The `SecurePurseClient` application uses a subclass of `CardAccessor` called `CustomCardAccessor` to perform user authentication functions and to sign every message sent thereafter for integrity purposes:

```
package examples.securepurseclient;

public class CustomCardAccessor extends ApduIOCardAccessor {
    /** Creates new CustomCardAccessor */
    public CustomCardAccessor() {
    }

    public byte[] exchangeAPDU(byte[] sendData) throws
java.io.IOException {

        byte[] macSignature = null;
        byte[] dataWithMAC = new byte[sendData.length + 4];

        // sign the sendData data using session key
        // sign the data in commandBuffer using the user's session key
        // add generated MAC signature to data in buffer before sending

        return super.exchangeAPDU(dataWithMAC);
    }

    boolean authenticateUser(short userKey) {
        byte[] externalAuthCommand = null;

        // build and send the appropriate commands to the
        // applet to authenticate the user using the user Key
        // and additional info provided
        try {
            byte[] response = super.exchangeAPDU(externalAuthCommand);
            // ...
        } catch (Exception e) {
            // analyze
            return false;
        }
        // Then compute the session key for later use
        return true; // successful authentication
    }
}
```

The `CustomCardAccessor` class introduces the `authenticateUser` method to send `APDU` commands to the `SecurePurseApplet` on the card to authenticate the user described by the `userKey` parameter and other parameters and to compute a transport key. It invokes `super.sendCommandAPDU` method to send the command without modification.

This `CustomCardAccessor` class also reimplements the `exchangeAPDU` method declared in a superclass `CardAccessor` to sign each message before it is sent out by `super.exchangeAPDU`.

Programming to the Java Card RMI Client-Side API

This section 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.
This section contains the following sections:

Overview of Programming to the Java Card RMI Client Side

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.

For best results use the Java Card RMI client-side API for Java Card RMI client programs. The simulator for the classic platform supports the optional Java Card RMI functionality.

The basic client-side framework is implemented in the package `com.sun.javacard.rmiclientlib and com.sun.javacard.clientlib`.

The library is located in the file `JC_HOME_TOOLS\lib\tools.jar`.

The simulator of the Java Card RMI client-side API is based on APDU I/O for its card access mechanisms. See Working with APDU I/O for more information on APDU I/O.

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.

You can use the standard Java RMIC compiler tool 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 *Java Card Platform Application Programming Interface Specification, Classic Edition, Version 3.1*.

> **Note:**
>
> Since the remote object is configured as a Java Card platform-specific object with a local connection to the smart card through 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 simulator demonstrate remote stub customization using the RMIC compiler generated stubs and card access for Java Card applets.

The package `com.sun.javacard.rmiclientlib` implements Java Card RMI-specific functionality.

The package `com.sun.javacard.clientlib` implements basic functionality to exchange APDUs with a smart card or a smart card simulator. This implementation of `clientlib` requires that the ApduIO library is included in the CLASSPATH.

Package rmiclientlib

This package includes several classes.

- **class JCRMIConnect**–The main class of the RMI framework that provides methods to select a card applet and to get an initial reference.
- **class JCCardObjectFactory**–An implementation of the `CardObjectFactory` that processes the data returned from the card in the format defined in the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.1*. Any object references must contain class names.
- **class JCCardProxyFactory**–The `JCCardProxyFactory` class is similar to `JCCardObjectFactory`, but processes references containing lists of names. `JCCardProxyFactory` uses the JDK 1.4.+ proxy mechanism to generate proxies dynamically.
- **class JCRemoteRefImpl**–An implementation of interface `java.rmi.server.RemoteRef`. These remote references can work with stubs generated by the RMIC compiler with the `-v1.2` option.

  The main method is: `public Object invoke(Remote remote, Method method, Object[] params, long unused) throws IOException, RemoteException, Exception`

This method prepares the outgoing APDU, passes it to `CardAccessor`, and then uses `CardObjectFactory` to parse the returned APDU and instantiate the returned object or throw an exception.

Package clientlib

This package includes an interface and a class.

- **interface CardAccessor**–An interface defining methods to exchange APDUs with a card and to close connection to a card.
- **class ApduIOCardAccessor**–A simple implementation of the CardAccessor interface that passes the APDUs to a card or a card simulator using the ApduIO library. This class takes parameters to start the ApduIO from the file `jcclient.properties`, which must be included in CLASSPATH.

14

Using Extended APDU

This chapter describes the Extended APDU and how it can be used to allow large amounts of data to be sent to the card, processed appropriately, and sent back to the terminal.
The Extended APDU feature is especially beneficial to applications that deal with large amounts of information, such as signature verification, biometrics verification and image storage and retrieval. These are more easily implemented if the underlying transport protocol is T=1. Applets developed for T=0 cards need special logic and care to work correctly.

This chapter includes the following topics:

Extended APDU Nominal Cases

The ISO/IEC 7816-4:2013 specification defines an extended APDU as any APDU whose payload data, response data or expected data length exceeds the 256 byte limit. Therefore, the four traditional cases are redefined as follows:

- Case 1. As in short length, this case is not affected.
- Case 2S. The legacy Case 2 from previous Java Card technology releases. LE has a value of 1 to 255.
- Case 2E. The extended version of Case 2S, where LE is greater than 255.
- Case 3S. The legacy Case 3. LC is less than 256 bytes of data, and LE is zero.
- Case 3E. The extended version of Case 3, where LC is greater than 255, and LE is zero.
- Case 4S. The legacy Case 4. LC and LE are less than 256 bytes of data.
- Case 4E. The extended version of Case 4. LC or LE are greater than 256 bytes of data.

Extended APDU Format

Any APDU classified as extended must follow the format defined by ISO/IEC 7816-4:2013 for extended length APDU and summarized in Table 14-1.

Table 14-1 Extended APDU Format

Field	Description	Number of Bytes
Command Header	Class byte CLA	1
Command Header	Instruction byte INS	1
Command Header	Parameter bytes P1- P2	2
LC Field	Absent for Nc = 0. Present for Nc > 0	0, 1, or 3
Data Field	Absent if Nc = 0, present if Nc >0	Nc
LE Field	Absent for Ne = 0, present for Ne > 0	0, 1, 2 or 3
Response Data	Absent if Nr = 0, present if Nr >0	Nr (max. Ne)
Response Status	Status bytes SW1 SW2	2

Notation

Nc = command data length

Ne = expected response data length

Nr = actual response data length

The encoding rules are defined as:

For LC:

- If LC field is absent, Nc = 0.
- If LC is present as one byte with values between `01` and `FF`, then Nc = 1..255 accordingly, and it will be a short field.
- If LC is present as an extended field, then it will be three bytes in length: byte one will be `00`, bytes two and three will contain a 16-bit value representing the length of the data Nc with values between 1 and 65535.

For LE:

- If LE is absent, Ne = 0.
- If LE is one byte:
  - A value between `01` and `FF` will indicate Ne = 1..255.
  - A value of `00` will indicate Ne = 256.

If LE is an extended field:

- LC and LE must be in the same format.
- An LE field value between `0001` and `FFFF` will indicate Ne = 1..65535.
- An LE field value of `0000` will indicate Ne= 65536.

Extended APDU Limits

The Java Card Platform, Version 3.1 supports extended APDUs with some limitations. Because the platform defines all of its mandatory API in terms of short data length, the values of LC and LE are limited to short positive values. That is, LC and LE have a range of `0..32,767`. Lengths of `32,768` and beyond are not supported by the Java Card Platform, Version 3.1 at this time.

This section includes the following topics:

- javacardx.framework.ExtendedLength Interface
- APDU Parsing with the javacard.framework.APDU Class

javacardx.framework.ExtendedLength Interface

By implementing the `javacardx.apdu.ExtendedLength` interface, applets indicate that they are capable of processing, receiving, and replying to extended APDU commands. The Java Card RE does not deliver extended APDU commands to applets that do not implement this interface (it would throw an `ISOException` with reason code `ISO7816.SW_WRONG_LENGTH`). In addition, the Java Card RE does not allow applets to send reply data lengths greater than 256, if the interface is not implemented by the applet.

The APDU buffer in Java Card applications reflects the structure of the extended APDU as defined in the ISO/IEC 7816-3 specification. In T=1, this representation is straightforward and precise; however, in T=0, adaptations are needed for some cases.

Specifically, a case 2E APDU sent over T=0 transport will not show its extended LE value in the APDU buffer. Instead, a P3 value of '00' will always be transmitted and interpreted as 32,767 if the applet implements `ExtendedLength`, or interpreted as 256 if it does not.

The Java Card RE analyzes the APDU type coming into the card and determines its type based on the rules defined in the ISO/IEC 7816-3 specification. Because case 2E commands look like case 2S commands in T=0, the Java Card RE is not able to distinguish this particular case.

APDU Parsing with the javacard.framework.APDU Class

Because LC in cases 3E and 4E can take a large value, the parameter is sent to the card as a three-byte quantity, in the format of `00 LCh LCl` starting at `ISO7816.OFFSET_LC`.

To get the value of LC and the data offset inside the APDU buffer use these two APIs in `javacard.framework.APDU`:

- `public short getIncomingLength()`

  This API call returns the value of LC as expressed in the APDU, whether it is extended or not.
- `public short getOffsetCdata()`

  This API call returns the offset where the first byte of the APDU data segment is found.

Creating an Applet That Can Send and Receive Extended Length APDUs

To create an applet that can send and receive extended length APDUs:

1. Implement the `javacardx.apdu.ExtendedLength` interface in your applet:

   ```
   ...
   import javacard.framework.*;
   import javacardx.apdu.ExtendedLength;
   ...
   public MyApplet extends Applet implements
   ExtendedLength {
   ...
   }
   ```

2. Write your applet and `Applet.process(..)` method as you would with any other applets. For consistency, it is advisable that your `process(..)` code begin like the one below:

   ```
   public void process(APDU apdu) {
      byte[] buffer = apdu.getBuffer();

      if (apdu.isISOInterindustryCLA()) {
          if (this.selectingApplet()) {
              return;
          } else {
              ISOException.throwIt (ISO7816.SW_CLA_NOT_SUPPORTED);
          }
      }

      switch (buffer[ISO7816.OFFSET_INS]) {
      case CHOICE_1:
          ...
          return;
      case CHOICE_2:
          ...
          ...
      default:
          ISOException.throwIt (ISO7816.SW_INS_NOT_SUPPORTED);
      }
   }
   ```

3. For cases 3S, 4S, 3E and 4E, write the method to handle incoming data. Use the API so that your applet properly handles extended, as well as non-extended, cases.

   ```
   void receiveData(APDU apdu) {
       byte[] buffer = apdu.getBuffer();
       short LC = apdu.getIncomingLength();

       short recvLen = apdu.setIncomingAndreceive();
       short dataOffset = apdu.getOffsetCdata();

       while (recvLen > 0) {
           ...
           [process data in buffer[dataOffset]...]
                   ...
   ```

```
                recvLen = apdu.receiveBytes(dataOffset);
    }
    // Done
}
```

4. For case 2S, 2E, write the method handling data output. A method could look something like this:

```
void sendData(APDU apdu) {
   byte[] buffer = apdu.getBuffer();

   short LE = apdu.setOutgoing();
   short toSend = ...

   if (LE != toSend) {
       apdu.setOutgoingLength(toSend);
   }

   while (toSend > 0) {
       ...
       [prepare data to send in APDU buffer]
               ...
               apdu.sendBytes(dataOffset, sentLen);
               toSend -= sentLen;
  }
  // Done
}
```

15 Working with APDU I/O

This chapter describes the APDU I/O API, which is a library used by many Java Card development kit components, such as `apdutool`, and the RMI client framework.

This chapter contains the following sections:

- The APDU I/O API
- Two-interface Card Simulation
- Examples of Use

The APDU I/O API

The APDU I/O library is used to develop Java Card client applications and Java Card platform simulators. It provides the means to exchange APDUs by using the T=0 or T=1 protocols.

The library is located in the file `JC_HOME_TOOLS\lib\tools.jar`.

All publicly available APDU I/O client classes are located in the package `com.sun.javacard.apduio`.

APDU I/O Classes and Interfaces

The APDU I/O classes and interfaces are described in this section.

- `class Apdu`

  Represents a pair of APDUs (both C-APDU and R-APDU). Contains various helper methods to access APDU contents and constants providing standard offsets within the APDU.

- `interface CadClientInterface`

  Represents an interface from the client to the card reader or a simulator. Includes methods for powering up, powering down and exchanging APDUs.

  – `void exchangeApdu(Apdu apdu)`

    Exchanges a single APDU with the card. Note that the APDU object contains both incoming and outgoing APDUs.

  – `public byte[] powerUp()`

    Powers up the card and returns ATR (Answer-To-Reset) bytes.

  – `void powerDown(boolean disconnect)`

    Powers down the card. The parameter, applicable only to communications with a simulator, means "close the socket". Normally, it is `true` for contacted connection, `false` for contactless. See Two-interface Card Simulation 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

The following exceptions may be thrown in case of system malfunction or protocol violations:

- `CadTransportException` extends `Exception`
- `T1Exception` extends `CadTransportException`
- `TLP224Exception` extends `CadTransportException`

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.

Two-interface Card Simulation

To simulate dual-interface cards with the simulator the following model is used:

- The simulator (`cref`) 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 (`cref`) 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.

APDU I/O API Examples

The following are examples of how to use the APDU I/O API:

- To Connect To a Simulator
- To Power Up And Power Down the Card
- To Exchange APDUs
- To Print the APDU

To Connect To a Simulator

To establish a connection to a simulator such as `cref`:

1. Use the following code snippet:

```
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);
```

2. The code establishes a T=0 connection to a simulator listening to port `9025` on `localhost`.

   To open a T=1 connection instead, replace `PROTOCOL_T0` in the last line with `PROTOCOL_T1`.

   For dual-interface simulation, open two T=1 connections on ports (*n*) and (*n*+1), as described in Two-interface Card Simulation.

To Power Up And Power Down the Card

The dual-interface simulator 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 power up and power down the card:

1. Use the following code:

   ```
   cad.powerUp();
   ```

2. 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();
   ```

3. To power down, but leave the socket open, use the following code.

   ```
   cad.powerDown(false);
   ```

   If the simulator continues to run (which is true if this is contactless interface of the simulator) you can issue `powerUp()` on this card again and continue exchanging APDUs.

To Exchange APDUs

To exchange APDUs:

1. Create a new APDU object using the following code:

   ```
   Apdu apdu = new Apdu();
   ```

2. Copy the header `(CLA, INS, P1, P2)` of the APDU to be sent into the `apdu.command` field.

3. 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.

4. Set the number of bytes expected into the `apdu.Le` field.

5. 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

To print the APDU:

- The following code prints both C-APDU and R-APDU in the `apdutool` output format.

  ```
  System.out.println(apdu)
  ```

16

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.
While the extended CAP files allow multiple packages in a single CAP file and method component of size greater than 64 K, the compact CAP files are still limited to a single package and have a 64 K limit on the method component. Therefore, you must take special considerations when using the compact CAP files to take advantage of the large memory storage in smart cards.

This chapter contains the following sections:

- Overview of Programming for the Large Address Space
- Programming Large Applications and Libraries
- Storing Large Amounts of Data
- Example: The photocard Demo Applet

Overview of Programming for the Large Address Space

The default address space automatically built in the simulator is the large address space. Allowing your applications to take advantage of the large address capabilities of the Classic Edition simulator using compact CAP files requires careful planning and programming. Some size limitations still exist within the simulator. The way that you structure large applications and applications that manage large amounts of data determines how the large address space can be exploited.

Programming Large Applications and Libraries

The introduction of Extended CAP files (see Programming Large Java Card Applications With Multiple Packages) in Java Card version 3.1 facilitates the development of large applications for the Java Card platform.

However, there might be scenarios where developers want to continue using the Compact CAP file format. For example, to target Java Card products that do not support Extended CAP files. When using the compact CAP file format, the most important limitation on a package is the 64 KB limitation on the maximum component size. This is especially true for the Method component. If the size of an application's Method component exceeds 64 KB, then the Java Card Converter doesn't process the package and returns 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. It is important to note 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 must not place sensitive or exclusive-use code in a library package. It must 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 lets 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 simulator 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 Platform simulator, enables 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 simulator enables 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 Programming to the Java Card RMI Client-Side API. For more information on the `photocard` demo applet and how to run it, see PhotoCard Sample. The source code is located in the source code bundle at:

```
JC_HOME_SIMULATOR\samples\classic_applets\PhotoCard\applet\src\c
om\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.

17 Programming Large Java Card Applications With Multiple Packages

The Java Card version 3.1 allows you to bundle multiple Java packages into one Java Card CAP file using an extended CAP file format.

This feature enables you to:

- Keep a modular design by having applications or libraries made of multiple packages.
- Distribute an application with the libraries it relies on.
- Control the visibility of each of the packages deployed in a CAP file.
- Overcome the size limitation of 64 K in compact CAP files.

The converter uses the extended CAP file format when multiple packages are used in a single CAP file or when the converted code for the whole CAP file exceeds the size of 64 KB.

The Java Card products optionally supports the extended CAP file. Before using this extended CAP file format, it is important to check if the target Java Card product supports this feature or not.

Refer to *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1*, for more information on the CAP file and its format.

Topics:

CAP File Identification

Each CAP file has an AID and a version. The AID and version values are dependent on the format of the CAP file (compact and extended) as follows:

- When the CAP file contains a unique package and it is in the compact format, its AID and version are same as the AID and version of the package it contains.
- When the CAP file contains multiple packages and it is in the extended format, it has its own AID and version, independent of the AIDs and versions of the packages it contains.

Package Visibility

The extended CAP file format offers more flexibility for the package visibility as follows:

- Each public package inside a CAP file has an AID and version. This AID and version uniquely identify this package when a package in another CAP file is importing it.
- Private packages inside a CAP file or packages that have no exported information (like an applet package with no `Shareable` interfaces) have no AID and version. In addition, because nothing is exported, the Converter does not generate an export file. Packages in other CAP files imports nothing from such packages.
- Unlike in an applet compact CAP file, if an extended CAP file is an applet CAP file, then the public library packages that are contained in the CAP file are exported as if they were present in a compact CAP file. In addition, the public applet packages contained in an extended CAP file are exported individually based on the same rules as for the compact CAP files (only public `Shareable` interfaces).
- Packages inside a bundle are visible to each other and the standard Java access rules (`public`, `protected`, `package`, or `private`) apply, irrespective of whether the packages are `public` or `private`.

Firewall Context

If a CAP file contains at least one package with one or more non-abstract classes that extend the `javacard.framework.Applet` class, then it is associated with a firewall context.

Refer to the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.1*, for more details.

Extended CAP Accessibility Example

To understand the extended CAP file accessibility, let’s consider a scenario as shown in the following figure:

The following table describes the package level access under different access conditions (1, 2, and 3):

Figure 17-1 Extended CAP Accessibility Example

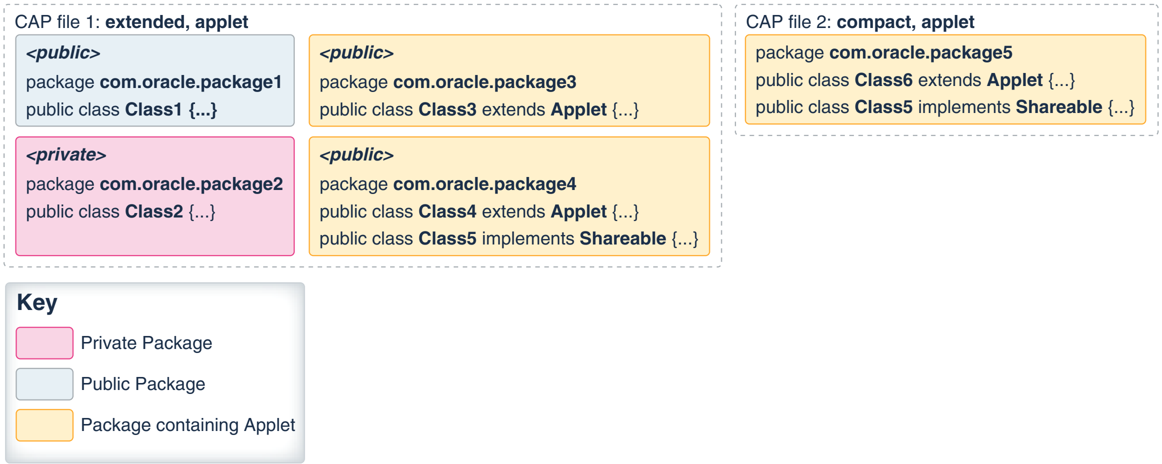

CAP file 1: extended, applet
<public>
package com.oracle.package1
public class Class1 {...}
<public>
package com.oracle.package3
public class Class3 extends Applet {...}
<private>
package com.oracle.package2
public class Class2 {...}
<public>
package com.oracle.package4
public class Class4 extends Applet {...}
public class Class5 implements Shareable {...}
CAP file 2: compact, applet
package com.oracle.package5
public class Class6 extends Applet {...}
public class Class5 implements Shareable {...}
Key
Private Package
Public Package
Package containing Applet

Table 17-1 Package Level Access

Accessibility	package1	package2	package3	package4	package5
package1 has access to:		Yes (1) and (2)	Yes (1)	Yes (1)	Yes (1), (2), and (3)
package2 has access to:	Yes (1) and (2)		Yes (1)	Yes (1)	Yes (1), (2), and (3)
package3 has access to:	Yes (1) and (2)	Yes (1) and (2)		Yes (1)	Yes (1), (2), and (3)
package4 has access to:	Yes (1) and (2)	Yes (1) and (2)	Yes (1)		Yes (1), (2), and (3)
package5 has access to:	Yes (1) and (2)	No (1)	No (1) and (3)	Yes (1) and (3)	

The following are the access conditions (1, 2, and 3) that are listed in the table:

1. **Exported packages in an extended CAP file** - Packages in an extended CAP file can be marked as `public` or `private`. Only the `public` packages are accessible from packages in another CAP file. However, all packages are accessible within the same CAP file.
2. **Java access rules** - Code access conforms to Java accessibility rules (`private`, `public`, `package`, `protected`, and so on.). For example, inside the `Class1` or `Class2` methods, a `Class3()`or a `Class4()` constructor can be called (only if the constructors are `public`) or other public methods from `Class3` and `Class4`, even if `Class3` and `Class4` are Java Card applets.
3. **Java Card access rules for package containing an Applet** - A public package containing a class extending the `javacard.framework.Applet` class does not export all its `public` classes and interfaces. Only the interfaces extending the `javacard.framework.Shareable` interface or the classes implementing it are exported and visible from other packages. For example, code from `package5` can access only the `Class5` in `package4` and cannot access content of `package3` because nothing is exported.

Design Rules for a Java Card Application with Large Method Component

In compact CAP files, the Method Component is limited to a 64 KB size. This can be a constraint if an application has many features, if a library has a large API, or if it is too large to fit into that size after conversion.

The extended CAP file offers a solution to this by creating a Method Component that has a maximum size of eight megabytes. For large applications, the extended mode is preferable.

The Converter splits the large Method Component into blocks, each with a maximum size of 64 KB. It is important to note that methods cannot be divided between two blocks and all exception handlers for a method must be contained in the same block of the method code. Because of this, when programming large Java Card applications for extended CAPs, the method code size must not be too large and specialization pattern must be used whenever possible.

18

Java Card Accessibility Information

Java Card Development Kit provides a wide range of features that support accessibility. Oracle is committed to creating products, services, and supporting documentation that is accessible to the disabled community. Java Card Runtime Environment is executed through command line interface.

This topic details the Java Card Development Kit features that support accessibility.

Topics:

- Access to Java Card Development Kit Support
- Java Card Development Kit Features that Support Accessibility
- Keyboard Navigation
- Documentation Accessibility Features

Access to Java Card Development Kit Support

The Java Card Development Kit customers have access to electronic support through email with their assigned support engineer provided by the Java Card Licensee Engineering (JLE) organization.

Hearing impaired customers in the U.S. who wish to speak to their assigned support engineer can use the telecommunications relay service (TRS). Information about the TRS is available at http://www.fcc.gov/cgb/consumerfacts/trs.html and a list of telephone numbers is available at https://www.fcc.gov/general/telecommunications-relay-services-directory International hearing-impaired customers must use the TRS at +1.605.224.1837.

Java Card Development Kit Features that Support Accessibility

Oracle's goal is to ensure that disabled users of our products can perform the same tasks, and access the same functionality as other users.

Java Card Development Kit supports the following accessibility features:

- Can be operated using only the keyboard
- Communicates all information independent of color
- Time Based Media is not used
- Images of text are not used
- Moving, blinking, or scrolling content is not used
- Doesn’t disrupt platform accessibility features such as Sticky Keys, High Contrast, and Large Fonts

- Provides online documentation in an accessible format

Keyboard Navigation

Java Card Development Kit uses standard navigation keys.

Documentation Accessibility Features

Java Card Development Kit documentation supports the following accessibility features:

- The documents are available in the HTML format to give maximum opportunity for the users to apply screen-reader software technology.
- The images in the documents are provided with alternative text so that users with vision impairments can understand the contents of the images.

Part III

Java Card Eclipse Plug-in

This part of the user guide describes how to use the Java Card Eclipse plug-in to create a Java Card project, Java Card applet, and to debug an applet. It contains the following chapter:

- Using the Java Card Eclipse Plug-in

19 Using the Java Card Eclipse Plug-in

This chapter contains the following topics:

Topics:

Creating a Java Card Project Using the New Java Card Project Wizard

To create a new Java Card Project, based on a default template, use the New Java Card Project wizard:

1. Click the **File** menu, and then select **New** and **Other…**.
2. In the Other… dialog, under **Oracle Java Card SDK**, select **Java Card Project**.
3. In the first page of the wizard, configure the following Java Card specific sections:
   - **Runtime Environment** - Select the Java Card Platform that you want to use in the project. The platform and devices are selected from the existing configuration. If you have not configured the platform and devices, you can use a link that opens the platform and devices settings page. If you are using the Oracle Java Card Simulator source bundle, an option to add the Java Card API source files to the project for API debugging is available.
   - **Java Card Tools** - Configure the Java Card tools bundle path, if not configured already.
   - **Application** - Configure a Java Card package and/or applet with names and AIDs. When an applet is configured, a default Applet source template is created.
4. Click **Finish**.

   A new Java Card project is created in the workspace containing the default CAP file configuration. However, this is valid only if the application is configured in Step 3. The API source files are added as linked sources in the build path if this option is selected in Step 3.

Changing the Runtime Environment for the Java Card Project

To change the runtime environment:

1. Right-click on the Java Card project and select **Java Card** and **Runtime Setting**.
2. Select a platform from the **Platform** section. If a platform is not configured, click the link that opens the Java Card platforms and device global settings pages.
3. In the **Device** section, select a configured device for the selected platform. If a device is not configured, click the link that opens the Java Card platforms and device global settings page.

Creating a Java Card Applet Using the Default Source Template

To create Java Card Applets use the default source template of the Eclipse plug-in:

1. Select a Java Card Project, click the **File** menu, and select **New** and **Other…**.
2. In the Other… dialog, expand **Oracle Java Card SDK** and select **Java Card Applet**.
3. In the **Package** section, click **Browse** and select the Java package where you want to add the Applet source template.
4. In the **Applet name** field, enter a Java class compliant name.
5. Click **Finish**.

   The Java Card applet is added to the package.

Creating a CAP File in a Java Card Project

To create deliverables in a Java Card project, you must create and configure CAP files.

To create a CAP file:

1. Select a Java Card Project, click the **File** menu, and select **New** and **Other…**.
2. In the Other… dialog, expand **Oracle Java Card SDK** and select the **Java Card CAP File**.
3. In the Select CAP file type page, select either **Compact CAP file** or **Extended CAP file**. Each type of the CAP file has a specific function in the Java Card specification. It is important to note the following:
   - A compact CAP file can have only one Java Card package configured.
   - An extended CAP file can be used only with the 3.1.0 or greater Java Card platform versions.
4. In the Select CAP file settings page, configure the `converter` and `scriptgen` tools options for the build:
   a. In the **CAP File AID** section, enter a unique CAP file name and CAP file AID. If the CAP file is a compact CAP file, then the AID field is disabled. This is

because, the CAP file AID is inherited from the Java Card package that is configured.

b. In the **Converter** section, configure the `converter` tool:

   - In the **Options** section, enter values for the CAP file version, target platform, and all the flags. If a CAP file is a compact CAP file, then the version is inherited from the Java Card package that is configured. If a CAP file is an extended CAP file, then the target platform must be 3.1.0 only and the mask flag option is disabled.
   - In the **Export Path** section, add the directories in which the `converter` tool searches for the export files. The `verifier` tool uses the paths added in this step during the build process. If the paths are added from the project, relative paths are generated.
   - In the **CAP Signing** section, configure the CAP sign feature of the `converter` tool.

c. In the **ScriptGen** section, specify the options for the `scriptgen` tool. For example, specify how to modify the default script templates for the CAP file loading and how to suppress `power up` and `power down` commands in the order in which you want to run the scripts.

5. Click **Finish**.

   A dialog appears prompting you to confirm if the Java Card package needs to be configured for the CAP file you just created. If you click **Yes**, the Package Configuration dialog appears. Else, if you click **No**, the wizard closes and a new configuration file to be included in the build is created.

Managing CAP File Configurations

To manage the CAP file configurations list and to edit a CAP file configurations, use the CAP file project settings:

1. Right-click on the Java Card project and select the **Java Card** and **CAP Files Settings**.
2. In the Java Card CAP Files page, to manage the CAP files used in the build, use the following options:

   a. Click **Add** to create a new CAP file using the wizard.

   b. Click **Edit** to edit a CAP file configuration using the wizard with all fields set to the previous values.

   c. Click **Delete** to delete a CAP file configuration from the build.

Adding a Java Card Package to a CAP File

To include a CAP file in the build, it must be in the configured state. A Configured CAP file is a CAP file that has at least one Java Card package added to it.

To add a Java Card package to a CAP file:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
2. In the Java Card CAP Files page, select a CAP file from the list.

3. Click **Add new package**.

   > **Note:**
   >
   > If the CAP file is a compact CAP file and it is already configured, then **Add new package** is disabled. A compact CAP file can contain a maximum of one package only.

4. In the Configure Java Card package dialog, configure the following `converter` tool specific parameters:
   a. Next to the **Package Name** field, click **Browse…** and select a Java package from the project

   b. Enter values for the Java Card package AID and version fields.

   c. Verify the `converter` tool flags. The **Private** flag is available only if the CAP files are extended CAP files.

5. Click **OK**.

   A dialog appears prompting you to confirm if a Java Card Applet needs to be configured for the package that you just created. If you click **Yes**, the Java Card Applet configuration dialog appears. Else, if you click **No**, the dialog closes, the CAP file configuration is updated, and the project is rebuilt.

Managing the Java Card Package

To manage the Java Card package added to a CAP file and to edit the Java Card package, use the CAP file project settings:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Setting**.
2. In the CAP Files Settings page, click the arrow to the left of the **Java Card CAP Files**.
3. Select **Java Card Packages**.

   A list with package names appears.
4. Select a CAP file from the combo list.

   The list is populated with the Java Card packages configured for the selected CAP file.
5. To manage the Java Card packages configured for the selected CAP file, perform the following tasks:
   a. Click **Add** to add a new Java Card package to the selected CAP file. If the CAP file is a configured compact file, this button is disabled.

   b. Click **Edit** to edit an already configured Java Card package.

   c. Click **Delete** to delete a Java Card package from the selected CAP file.

Adding a Java Card Applet to a Java Card Package

If a CAP file is an applet CAP file, then you need to configure the applets that are contained in it. A `Java Card Applet` is a class contained in a Java Card package.

To add a Java Card Applet to a Java Card package:

1. Right-click on the Java Card project and select the **Java Card** and **CAP Files Settings**.
2. In the CAP Files Settings page, click the arrow to the left of the **Java Card CAP Files**.
3. Select **Java Card Packages** and the CAP file in which you want to add the Applet.
4. Select the package that contains the Applet you want to add.
5. Click **Add new applet**.
6. In the Configure Java Card Applet window, set the `converter` tool parameters for the applets:
    1. In the **Applet** section, click **Browse…** and select the applet class from the list.
    2. In the **Applet AID** section, enter a value for the PIX part of the AID of the Java Card Applet. The RID part of the AID is automatically populated based on the Java Card package configuration.
7. Click **OK**.

    The window closes, the CAP file configuration is updated, and the project is rebuilt.

Managing Java Card Applets

To manage Java Card Applets and to edit them, use the CAP file project settings:

1. Right-click on the Java Card project and select the **Java Card** and **CAP Files Settings**.
2. In the **CAP Files Settings** page, click the arrow to the left of the **Java Card CAP Files**.
3. Click the arrow to the left of the **Java Card Packages**.
4. Select **Java Card Applets**.

    A list with Applets appears.
5. Select a CAP file and Java Card package combination from the combo list.

    The list is populated with the Java Card Applets configured for the selected combination.
6. To manage the Java Card Applets configured for the selected combination, perform the following tasks:
    1. Click **Add** to add a new Java Card Applet to the selected CAP file or package combination.
    2. Click **Edit** to edit an already configured Java Card Applet.
    3. Click **Delete** to delete a Java Card Applet from the selected CAP file or package combination.

Adding a Java Card Static Resource to a CAP File

Since the Java Card version 3.1.0 and later, you can add static resources to a CAP file while loading.

To add a Java Card static resource to a CAP file:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
2. In the Java Card CAP Files page, select a CAP file from the list.
3. Click **Add new static resource**.
4. In the **Configure Java Card static resource** dialog, configure the `converter` tool static resource-specific parameters:
   a. In the **Static Resource ID field**, enter a unique integer number.
   b. In the **Static Resource file path** section, click **Browse…** and select the file that you want to add as a static resource to a CAP file. If the path is inside the project, a relative path is generated.
5. Click **OK**.

   The **Configure Java Card static resource** dialog closes, the CAP file configuration is updated, and the project is rebuilt.

Managing Java Card Static Resources

To manage Java Card static resources list added to a CAP file, and to edit the Java Card static resource, use the CAP file project settings:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Setting**.
2. In the CAP Files Settings page, click the arrow to the left of the **Java Card CAP Files**.
3. Select **Java Card Static Resources**.

   A list with static resources appears.
4. Select a CAP file from the combo list.

   The list is populated with Java Card static resources configured for the selected CAP file.
5. To manage the Java Card static resources configured for the selected CAP file, perform the following tasks:
   a. Click **Add** to add a new Java Card static resource to the selected CAP file.
   b. Click **Edit** to edit an already configured Java Card static resource.
   c. Click **Delete** to delete a Java Card static resource from the selected CAP file.

Debugging a Java Card Applet in Eclipse Plug-in

From Eclipse, you can run the debug proxy to set breakpoints, get or set variable values, and debug a library.

These steps are an overview of how to debug an application from Eclipse.

The Java Card plug-in for Eclipse must already be installed.

1. Create (or import) your Java Card project. Make sure that debugging information is generated when the project is built.

> **Note:**
>
> The debugging information is generated only if you select the **Enable generation of debugging information** check box in the CAP file settings.

2. Create a new Java Card Project Debug configuration and select the **Java Card** tab. Perform the following tasks:
   a. Specify scripts to be executed when the simulator starts. This not only includes the scripts generated in the `apdu_scripts` directory, but also other custom scripts. The scripts run in an order. The `powerup` command is sent only once at the beginning and the `powerDown` command is not sent. It is important to note that the `cap-*` scripts for libraries must be executed before the `cap-*` scripts for the applet if the EEPROM of `cref` is empty at the start of the debug session.
   b. Add cap files for the applet and imported libraries, which include the code that you want to debug. These not only include CAP files generated in the `deliverables` directory, but also the CAP files that you generated.
3. Once the debug session starts, `cref` starts in debug mode, the script(s) are executed, the debug proxy is started, and the Eclipse debugger connects to the debug proxy.

   You can experiment with the debug perspective and look at the debug console for debug proxy output.
4. Set breakpoints, and execute scripts.

Debugging HelloWorld Sample from Eclipse

These steps show you how to debug the `HelloWorld` sample. The Java Card plug-in for Eclipse must already be installed.

Start Eclipse. Sample_Platform and Sample_Device must already be created.

1. Click the **File** menu, select **Import >General > Projects from Folder or Archive**, and select the `applet` directory from the HelloWorld project to import the HelloWorld Java Card project into your workspace. If the build doesn't start automatically, start it manually.
2. Make sure debugging information generation is enabled for the `HelloWorld` package:
   a. Right-click on the imported project and select **Java Card** and **CAP Files Settings**.
   b. Select the `HelloWorld` CAP file and click **Edit**.
   c. In the new wizard, click **Next** on the first page.
   d. In the second page, select the **Enable generation of debugging information** check box.
   e. Click **Finish** and **Apply**, and close the wizard.
3. Create a new debug configuration:
   a. Right-click on the `HelloWorld` project in the Package Explorer and select **Debug As** and **Debug Configurations**.

   b. In the Debug Configurations dialog, double-click **Java Card Project Debug** (in the list). This will a create new debug configuration named HelloWorld.

   c. Select the **Java Card** tab.

   d. Select the **Start simulator in debug mode…** and **Start debug proxy…** check boxes.

   e. Click **Add script...**. Browse to the `HelloWorld` project directory and select the `applet\apdu_scripts\cap-HelloWorld.script` file. This script will install the applet without creating an applet instance.

   f. Click **Add cap file...**. Browse to the `HelloWorld` project directory and select the `applet\deliverables\HelloWorld\com\oracle\jcclassic\samples\helloworld\javacard\helloworld.cap` file.

   g. Click **Debug**.

   The debug configuration starts. First, `cref` is started in debug mode, then the script is executed, the debug proxy is started, and finally the Eclipse debugger connects to the debug proxy.

4. The Confirm Perspective Switch dialog appears, asking if you want to open the Debug perspective. You may choose to open it, depending on your preference.

   The Debug console shows output from the debug proxy.

5. In the Package Explorer, locate `HelloWorld.java` and open it. Set two breakpoints: one in the `install()` method of the applet, the other in the beginning of the `process()` method.

   There are several ways to set a breakpoint in Eclipse. In the source code editor, position the cursor on the desired line and do one of the following:

   a. Double-click the left most space on the source code line (the line number will be to the right).

   b. Press **Ctrl + Shift + B** to toggle the breakpoint (the type of breakpoint will be selected automatically depending on the source code).

   c. Select a specific breakpoint to toggle from the **Run** menu.

6. Execute the two remaining scripts in order they appear in the Package Explorer: `create-*`, then `select-*` (Right-click on the script and select **Java Card** and **Execute Script**).

   After each script runs, execution will suspend on the corresponding breakpoint. `Step*` and `resume` debugger commands can be used to resume applet code execution.

Part IV
Appendices

The following appendices contain a Java Card assembly syntax example and a description of additional, optional Ant tasks:

- Java Card Assembly Syntax Example
- Additional Optional Ant Tasks

A

Java Card Assembly Syntax Example

This appendix contains two examples of annotated Java Card platform assembly (Java Card Assembly) files that are generated with the Converter. The first example contains the output of 3.0.5 Converter. The second example highlights the changes in Java Card Assembly (JCA) files generated with the 3.1 Converter. A notable change is that the `publicMethodTable` format has changed due to the Virtual Method Tokens feature (see Section 6.9.2.7 in Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.1, for more details). The comments in these files are intended to help you understand the syntax of the Java Card Assembly language, and to act as a guide for debugging the Converter output.

> **Note:**
>
> If you are using a source release, you can get an HTML file with the BNF grammar for the Java Card Assembly syntax by using the Java `jjdoc` tool with:
>
> *`JC_HOME_TOOLS`*`\src\tools\converter\com\sun\javacard\jcasm\Parser.jj`

```
/*

 * 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;
l0:
l1:
l2:
l3:
l4:
l5:
            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;
                    l0 l4 l5 3;
                    l1 l3 l5 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 3.0.5.
             * Mnemonics are case insensitive.
             *
             * See the Java Card virtual machine specification for
             * the specification of
             * each instruction.
             */
            .method public instructions()V 4 {
                aaload;
                aastore;
                aconst_null;
       aload 0;
       aload_0;
       aload_1;
       aload_2;
       aload_3;
       anewarray 0;
```

```
areturn;
arraylength;
astore 0;
astore_0;
astore_1;
astore_2;
astore_3;
athrow;
baload;
bastore;
bipush 0;
bspush 0;
checkcast 10 0;
checkcast 11 0;
checkcast 12 0;
checkcast 13 0;
checkcast 14 0;
dup2;
dup;
dup_x 0x11;
getfield_a 1;
getfield_a_this 1;
getfield_a_w 1;
getfield_b 1;
getfield_b_this 1;
getfield_b_w 1;
getfield_i 1;
getfield_i_this 1;
getfield_i_w 1;
getfield_s 1;
getfield_s_this 1;
getfield_s_w 1;
getstatic_a 4;
getstatic_b 4;
getstatic_i 4;
getstatic_s 4;
goto 0;
goto_w 0;
i2b;
i2s;
iadd;
iaload;
iand;
iastore;
icmp;
iconst_0;
iconst_1;
iconst_2;
iconst_3;
iconst_4;
iconst_5;
iconst_m1;
idiv;
if_acmpeq 0;
if_acmpeq_w 0;
if_acmpne 0;
if_acmpne_w 0;
if_scmpeq 0;
if_scmpeq_w 0;
if_scmpge 0;
if_scmpge_w 0;
```

```
if_scmpgt 0;
if_scmpgt_w 0;
if_scmple 0;
if_scmple_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 {
    }
}


// converted by version  [v3.1.0]
.package package1 {
    .aid 0x1:0x2:0x3:0x4:0x5:0x1;
    .version 1.1;

    .imports {
        0xA0:0x0:0x0:0x0:0x62:0x0:0x1 1.0;     //java/lang
    }

    .constantPool {
        // 0
        staticMethodRef 0.0.0()V;        // java/lang/Object.<init>()V
    }

    .class public Class1 0 extends 0.0 {        // extends java/lang/Object

        .publicMethodTable 1 3 {   // 3 is 
CAP22_inheritable_public_method_token_count, see 3.1 JCVMSpec 6.9.2.7
            equals(Ljava/lang/Object;)Z 0;
            m1()S 255;  //m1 defined in Class1 1.0 and 1.1
            m2()S 255;  //m2 defined in Class1 1.0 and 1.1
            m4()S 255;  //m4 defined in Class1 1.1
        }

        .packageMethodTable 0 {
        }

        .method public <init>()V 0 {
            .stack 1;
            .locals 0;

                L0:    aload_0;
                    invokespecial 0;        // java/lang/Object.<init>()V
                    return;
        }

        .method public m1()S 1 {
            .stack 1;
            .locals 0;

                L0:    sspush 170;
                    sreturn;
        }

        .method public m2()S 2 {
            .stack 1;
            .locals 0;

                L0:    bspush 55;
                    sreturn;
        }

        .method public m4()S 3 {
            .stack 1;
            .locals 0;

                L0:    sspush 221;
                    sreturn;
        }

    }

}

// converted by version  [v3.1.0]
.package package2 {
    .aid 0x1:0x2:0x3:0x4:0x5:0x2;
    .version 1.1;

    .imports {
        0x1:0x2:0x3:0x4:0x5:0x1 1.1;        //package1
        0xA0:0x0:0x0:0x0:0x62:0x0:0x1 1.0;        //java/lang
    }

    .constantPool {
        // 0
        staticMethodRef 0.0.0()V;        // package1/Class1.<init>()V
    }

    .class public Class2 0 extends 0.0 {        // extends package1/Class1

        .publicMethodTable 2 4 {   // 4 is
CAP22_inheritable_public_method_token_count, see 3.1 JCVMSpec 6.9.2.7
            equals(Ljava/lang/Object;)Z 0;
            m1()S 1;     //inherited from Class1 1.0
            m2()S 2;     //method overridden in Class2 1.1
            m3()S 255;   //defined in Class2 1.0
            m4()S 3;     //inherited from Class1 1.1
            m5()S 255;   //defined in Class2 1.1
        }

        .packageMethodTable 0 {
        }

        .method public <init>()V 0 {
            .stack 1;
            .locals 0;

                L0:    aload_0;
                    invokespecial 0;        // package1/Class1.<init>()V
                    return;
        }

        .method public m2()S 2 {
            .stack 1;
            .locals 0;

                L0:    sspush 187;
                    sreturn;
        }

        .method public m3()S 3 {
            .stack 1;
            .locals 0;

                L0:    sspush 204;
                    sreturn;
        }

        .method public m5()S 5 {
            .stack 1;
            .locals 0;

                L0:    sspush 238;
                    sreturn;
        }

    }

}
```

B

Additional Optional Ant Tasks

This appendix contains a description of the optional Ant tasks supported by this development kit. 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. 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.

This chapter includes the following sections:

- Location and Installation
- Setting Up the Optional Ant Tasks
- Ant Task Descriptions
- Custom Types

Location and Installation

The optional Ant tasks are included at:

- *JC_HOME_TOOLS*`\lib\jctasks_tools.jar`
- *JC_HOME_SIMULATOR*`\lib\jctasks_simulator.jar`

> **Note:**
>
> Use of the additional Ant tasks described in this section is strictly optional and is not formally supported, nor has it been fully tested.

Installing the Ant Tasks

1. Be sure Ant is configured as described in Downloading the Development Kit.
2. Copy the file *JC_HOME_TOOLS*`\lib\jctasks_tools.jar` or *JC_HOME_SIMULATOR*`\lib\jctasks_simulator.jar` to a directory that serves as your Ant tasks home directory.
3. Add the `jctasks_tools.jar` or `jctasks_simulator.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:

   a. *Ant-Home-Path* is the path to the Ant installation.

b. The value of the ANT_HOME environment variable is properly configured to run Ant (see Downloading the Development Kit).

Setting Up the Optional Ant Tasks

The following XML must be added to 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="capdump"
  classname="com.sun.javacard.ant.tasks.CapdumpTask" />
<taskdef name="capgen"
  classname="com.sun.javacard.ant.tasks.CapgenTask" />
<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 JAR files located in *JC_HOME_SIMULATOR*\lib\tools_simulator.jar and *JC_HOME_TOOLS*\lib\tools.jar contain the libraries required to execute the optional ant tasks. These JAR files must be in the classpath during build execution.

Ant Task Descriptions

The 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
- CapDump
- Capgen
- Converter
- DeployCap

- Exp2Text
- Scriptgen
- VerifyCap
- VerifyExp
- VerifyRev

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 Machine[1] software (JVM software) than the one being used by Ant.

Table B-1 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 `cref` executable.	Yes
OutEEFile	Output EEPROM file that contains the EEPROM image after `cref` finishes execution.	Yes
CheckDownload Failure	Stops the build if any response status coming back from `cref` 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 `cref`. If specified `cref` initiates using the EEPROM image stored in this file.	No
nobanner	Set this element to `true` 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 the required attributes are not provided, *JC_HOME_SIMULATOR*`\lib\tools_simulator.jar` and *JC_HOME_TOOLS*`\lib\tools.jar` are not in the classpath, or APDUTool returns an error code.

Examples

To use these examples:

[1] The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java(TM) platform.

1. Enter the following example code to run 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>
```

2. Enter the following example code to run the APDUTool to install the APDU script in `test.scr` file to `cref` and check if the APDU commands were processed successfully:

   Note:

   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>
```

3. Enter the following example code to 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`:

   Note:

   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

Run the CapDump tool to dump the contents of a CAP file.

Table B-2 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 attribute is not supplied, *JC_HOME_TOOLS*`\lib\tools.jar` is not in the classpath, or CapDump returns an error code.

Examples

To use these examples:

1. Enter the following example code to run CapDump to dump the contents of the `test.cap` file:

```
<target name="CapDumpTarget" >
  <capdump>
      CapFile="${samples.output}/test.cap"
  </capdump>
</target>
```

2. Enter the following example code to run CapDump to dump the contents of the `test.cap` file:

> **Note:**
>
> Classpath used in this example code 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-3 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 `true` if you do not want the Capgen banner showing.	No
version	Prints Capgen version number.	No

Errors

Execution of this task fails if the required attributes are not provided, *JC_HOME_TOOLS*`\lib\tools.jar` is not in the classpath, or Capgen returns an error code.

Examples

To use these examples:

1. Enter the following example code to run Capgen to generate the `helloworld.cap` file from the `helloworld.jca` file.

```
<target name="CapgenTarget" >
  <capgen
      JCAFile="${sample.output}/helloworld.jca"
      outfile="${sample.output}/helloworld.cap">
      </capgen>
</target>
```

2. Enter the following example code to run Capgen to generate a `helloworld.cap` file from the `helloworld.jca` file.

> **Note:**
>
> Classpath used in this example is referenced by the `classpath` refid.

```
<target name="CapgenTarget" >
  <capgen
      JCAFile="${sample.output}/helloworld.jca"
      outfile="${sample.output}/helloworld.cap">
      <classpath refid="classpath"/>
  </capgen>
</target>
```

3. Enter the following example code to run Capgen as in the previous example, except no output file is specified.

> **Note:**
>
> Capgen generates `out.cap` in the directory in which the JVM software was invoked.

```
<target name="CapgenTarget" >
  <capgen
      JCAFile="${sample.output}/helloworld.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-4   Parameters for Converter

Attribute	Description	Required
PackageName	Fully qualified name of the package being converted.	Yes, if the configuration file is not provided.
PackageAID	AID of the package being converted.	Yes, if the configuration file is not provided.
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, if the configuration file is not provided.
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 looks for class files.	No
Int	If enabled, turns on support for the 32-bit integer type.	No
Debug	If enabled, enables generation of debugging information.	No

Table B-4 (Cont.) Parameters for Converter

Attribute	Description	Required
ExportPath	Root directories where the Converter looks 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 looks for the export file in the exportpath.	No
Outputdirectory	Sets the output directory where the output files are 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
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 Capgen banner showing.	No
version	Prints Converter version number.	No
ConfigFile	Configuration file containing all the configuration parameters for the converter.	No

In addition to the parameters specified in the preceding table, the target Java Card platform can be specified for the converter through the environment variable `JC_TARGET_PLATFORM`. If this environment variable is set, then the converter creates the CAP files for the specified target platform.

Parameters Specified As Nested Elements

The **AppletNameID** parameters are specified as nested elements and use nested element AppletNameAID to specify names and AIDs of applets belonging to the package being converted. For details regarding AppletNameAID type, see AppletNameAID .

Errors

Execution of this task fails if the required attributes are not provided, *JC_HOME_TOOLS*`\lib\tools.jar` is not in the classpath, or the Converter returns an error code.

Examples

To use these examples:

1. Enter the following example code to run the Converter and 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>
```

2. Enter the following example code to run the Converter with the converter options specified in the `helloworld.cfg` file instead of being specified in the target itself.

> **Note:**
>
> 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-5 Parameters for DeployCap

Attribute	Description	Required
CapFile	Fully qualified path and name of the CAP file which is to be sent to `cref`.	Yes
CrefExe	Fully qualified path and name of `cref` executable.	Yes
OutEEFile	Output EEPROM file that contains the EEPROM image after `cref` finishes execution.	Yes
InEEFile	Input EEPROM file for `cref`. If specified `cref` 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 `true` if you do not want the tool banner showing.	No

Errors and Return Codes

Execution of this task fails if the required attributes are not provided, *JC_HOME_SIMULATOR*`\lib\tools_simulator.jar` and *JC_HOME_TOOLS*`\lib\tools.jar` are not in the classpath, APDUTool, `scriptgen`, or `cref` fail to execute.

Examples

To use these examples:

1. Enter the following example code to install `helloworld.cap` file in `cref`:

> **Note:**
>
> 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>
```

2. Enter the following example code to install `helloworld.cap` file in `cref`, which in this case is initialized with EEFile:

> **Note:**
>
> The `cref` output EEPROM image is also 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-6   Parameters for Exp2Text

Attribute	Description	Required
PackageName	Fully qualified name of the package.	Yes
ClassDir	Root directory where the exp2text tool looks 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 the required attributes are not provided, *JC_HOME_TOOLS*`\lib\tools.jar` is not in the classpath, or `Exp2Text` returns an error code.

Examples

To use these examples:

1. Enter the following example code to run `Exp2Text` and generate text file from the export file of package HelloWorld:

> **Note:**
>
> This example assumes that `converter.jar` is already in classpath.

```
<target name="Exp2TextTarget" >
  <exp2text
      packagename="com.oracle.jcclassic.samples.helloworld"
      classdir="${classroot}"
      outputdir="${classroot}">
  </exp2text>
</target>
```

2. Enter the following example code to run `Exp2Text` and generate text file from the export file of package HelloWorld:

> **Note:**
>
> `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.oracle.jcclassic.samples.helloworld">
      <classpath refid="classpath"/>
  </exp2text>
</target>
```

Scriptgen

Runs Scriptgen to generate an APDU script file from a CAP file.

Table B-7 Parameters for Scriptgen

Attribute	Description	Required
CapFile	Fully qualified path and name of the input CAP file.	Yes
HashFile	Fully qualified path and name of the verifier-generated file that contains the hashes for all the components in 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 is 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

Table B-7 (Cont.) Parameters for Scriptgen

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 `true` if you do not want the Scriptgen banner showing.	No
version	Prints Scriptgen version number.	No

Errors

Execution of this task fails if the required attributes are not provided, *JC_HOME_SIMULATOR*`\lib\tools_simulator.jar` and *JC_HOME_TOOLS*`\lib\tools.jar` are not in the classpath, or Scriptgen returns an error code.

Examples

To use these examples:

- Enter the following example code to run Scriptgen and generate script file `helloWorld.scr` from `helloWorld.cap` file.

> **Note:**
>
> Classpath used in this example is referenced by the `classpath` refid.

```
<target name="ScriptgenTarget" >
  <scriptgen
      noBeginEnd="true"
      noBanner="true"
      HashFile="${samples.helloworld.output}/HelloWorld.hash"
      CapFile="${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-8 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 verifier 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
outFile	Fully qualified output path of the digest file, which contains the digests generated using the default algorithm (SHA-256) for all CAP file components.	
nobanner	Set this element to `true` if you want to suppress Verifier banner.	No
version	Prints the version number of the off-card verifier.	No

Parameters Specified As Nested Elements

The ExportFiles are parameters specified as nested elements that 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.

Errors

Execution of this task fails if the required attributes are not provided, *JC_HOME_TOOLS*`\lib\tools.jar` is not in the classpath, or Verifier returns an error code.

Examples

To use these examples:

- Enter the following example code to run the Java Card platform off-card verifier and verify the `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-9 Parameters for VerifyExp

Attribute	Description	Required
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose verifier 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 want to suppress Verifier banner.	No
version	Prints the version number of off-card verifier.	No

Parameters Specified As Nested Elements

The ExportFiles are parameters specified as nested elements that use nested element ExportFiles to specify the EXP file being verified. For details regarding ExportFiles type see ExportFiles. VerifiyExp requires that only one input EXP file be specified. This tasks throws an error if more than one EXP files are specified.

Errors

Execution of this task fails if none or more than one EXP file are specified, *JC_HOME_TOOLS*`\lib\tools.jar` is not in the classpath, or Verifier returns an error code.

Examples

To use these examples:

- Enter the following example code to run the Java Card platform off-card verifier to verify `HelloWorld.exp` file:

```
<target name="VerifyExpTarget" >
  <verifyexport>
      <exportfiles file="${samples.helloworld.output}/HelloWorld.exp" />
      <classpath refid="classpath"/>
  </verifyexport>
</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-10 Parameters for VerifyRev

Attribute	Description	Required
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose verifier 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 want to suppress Verifier banner.	No
version	Prints the version number of off-card verifier.	No

Parameters Specified As Nested Elements

The ExportFiles are parameters specified as nested elements that use nested element ExportFiles to specify the EXP files being verified. For details regarding ExportFiles type see ExportFiles. VerifyExp requires that exactly two input EXP files are specified: it throws an error if that is not the case.

Errors

Execution of this task fails if no EXP file is specified. It also fails if exactly two EXP files are not specified, *JC_HOME_TOOLS*`\lib\tools.jar` is not in the classpath, or Verifier returns an error code.

Examples

To use these examples:

- Enter the following example code to run the Java Card platform off-card verifier to verify binary compatibility between two versions of `HelloWorld.exp` file.

```
<target name="VerifyExpTarget" >
  <verifyrevision>
      <exportfiles file="${samples.helloworld.output}/HelloWorld.exp" />
      <exportfiles file="${samples.helloworld.output.new}/HelloWorld.exp" />
      <classpath refid="classpath"/>
  </verifyrevision>
</target>
```

Custom Types

This section includes the following information and description about available custom types:

- AppletNameAID
- JCAInputFile
- ExportFiles

AppletNameAID

AppletNameAID groups together name and AID for a Java Card applet.

Table B-11 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

To use these examples:

- Enter the following example code to 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-12 Parameters for JCAInputFile

Attribute	Description	Required
inputfile	Fully qualified name of the input file	Yes

Examples

To use these examples:

1. Enter the following example code to set the fully qualified name of an input JCA file.

   ```
   <jcainputfile
      inputfile="C:\jcas\common\com\sun\javacard\installer
   \javacard\installer.jca" />
   ```

2. Enter the following example code to set the fully qualified name of an input file that contains a list of JCA files.

   ```
   <jcainputfile inputfile="@C:\jc\jcaDemo.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

To use these examples:

1. Enter the following example code to set the fully qualified name of an input EXP file:

   ```
   <exportfiles
      file="C:\samples\classes\com\sun\javacard\samples
   \HelloWorld\javacard\HelloWorld.exp" />
   ```

2. Enter the following example code to group 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>
   ```

Glossary

active applet instance

an applet instance that is selected on at least one of the logical channels.

AID (application identifier)

defined by ISO 7816, a string used to uniquely identify card 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 11byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies.

A unique AID is assigned to each CAP file and public packages in a CAP file. In addition, a unique AID is assigned to each applet in the CAP file. The AID for the CAP file, the package AID of every public package in a CAP file, and the default AID for each applet defined in the CAP file are specified. They are supplied to the converter when the CAP file is generated.

APDU

an acronym for Application Protocol Data Unit as defined in ISO 7816-4.

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 is the basic unit of selection, context, functionality, and security in the Java Card technology.

applet application

an application that consists of one or more applets.

applet framework

an API that enables applet applications to be built.

applet developer

a person creating an applet using Java Card technology.

applet execution context

currently active applet owner identifier.

applet firewall

the mechanism that prevents unauthorized accesses to objects in contexts other than currently active context.

applet CAP file

a CAP file that contains one or more applet packages. See applet package.

applet package

a Java programming language package that contains one or more non-abstract classes that extend the `javacard.framework.Applet` class. See also library package.

assigned logical channel

the logical channel on which the applet instance is either the active applet instance or will become the active applet instance.

atomic operation

an operation that either completes in its entirety or no part of the operation completes at all.

atomicity

state in which a particular operation is atomic. Atomicity of data updates guarantee that data are not corrupted in case of power loss or card removal.

ATR

an acronym for Answer to Reset. An ATR is a string of bytes sent by the Java Card platform after a reset condition.

authentication

the process of establishing or confirming an application or a user as authentic using some sort of credentials.

basic logical channel

logical channel 0, the only channel that is active at card reset in the APDU application environment. This channel is permanent and can never be closed.

big-endian

a technique of storing multibyte data where the high-order bytes come first. For example, given an 8-bit data item stored in big-endian order, the first bit read is considered the high bit.

binary compatibility

in a Java Card system, a change in a Java programming language package in a Java Card CAP file results in a new CAP file. A new CAP file is binary compatible with (equivalently, does not break compatibility with) a preexisting CAP file if another CAP file converted using the export files of the packages included in the preexisting CAP file can link with the new CAP file without errors.

bytecode

machine-independent code generated by the compiler and executed by the Java virtual machine.

CAD

an acronym for Card Acceptance Device. The CAD is the device in which the card is inserted.

CAP file

Standard file format containing a binary representation of a shared library (library CAP file) or an application with its libraries that might be exported or not (applet CAP file).

A CAP file represents a module, which is a unit of code, made of one or more Java packages, with dependencies and list of exported packages and an assigned name (AID) for lifecycle management. Its structure is made of multiple CAP components deployed within a JAR file

When a CAP file containing application(s) is deployed on a Java Card platform, it is assigned a new unique group context that must be associated with any application instance created from code within this application module.

CAP file component

A Java Card platform CAP file consists of a set of components, which represent a set of one or more Java programming language packages. Each component describes a set of elements or an aspect of the CAP file. A complete CAP file must contain all of the required components: Header, Directory, Import, Constant Pool, Method, Static Field, and Reference Location.

The following components are conditionally included or optional: the Applet, Export, Static Resources and Debug. The Applet component is included only if one or more Applets are defined in one or more packages in the CAP file. The Export component is included only if one or more packages are public and exported allowing classes in other packages to import elements from them. The Static Resources component is included only if static resources are embedded in the CAP file. The Debug component is optional. It contains all of the data necessary for debugging.

cast

the explicit conversion from one data type to another.

card session

a card session begins when it is powered up or reset. The card is then able to exchange messages with external clients. The card session ends when the card loses power or is reset.

client application

an on-card application that uses services provided by other applications (server applications).

constant pool

the constant pool contains variable-length structures representing various string constants, class names, field names, and other constants referred to within the CAP file and the Export File structure. Each of the constant pool entries, including entry zero, is a variable-length structure whose format is indicated by its first tag byte. There are no ordering constraints on entries in the constant pool. One constant pool is associated with each package.

There are differences between the Java platform constant pool and the Java Card technology-based constant pool. For example, in the Java platform constant pool there is one constant type for method references, while in the Java Card constant pool, there are three constant types for method references. The additional information provided by a constant type in Java Card technologies simplifies resolution of references.

context

protected object space associated with each applet `CAP` file and Java Card RE. All objects owned by an applet belong to the context associated with the applet's `CAP` file.

context switch

a change from one currently active context to another. For example, a context switch is caused by an attempt to access an object that belongs to an application instance that resides in a different application group. The result of a context switch is a new currently active context.

converter

a piece of software that preprocesses all of the Java programming language class files contained in a set of packages and converts them into a CAP file. The Converter also produces *export files* for exported packages.

currently active context

when an object instance method is invoked, an owning context of the object becomes the currently active context.

currently selected applet

the Java Card RE keeps track of the currently selected Java Card applet. Upon receiving a `SELECT FILE` command with this applet's AID, the Java Card RE makes this applet the currently selected applet. The Java Card RE sends all APDU commands to the currently selected applet.

custom CAP file component

a new component added to the `CAP` file. The new component must conform to the general component format. It is silently ignored by a Java Card virtual machine that does not recognize the component. The identifiers associated with the new component are recorded in the `custom_component` item of the `CAP` file's Directory component.

default applet

an applet that is selected by default on a logical channel in the APDU application environment when it is opened. If an applet is designated the default applet on a particular logical channel in the APDU application environment on the Java Card platform, it becomes the active applet by default when that logical channel is opened using the basic channel.

EEPROM

an acronym for Electrically Erasable, Programmable Read Only Memory.

entry point method

well-defined method of an object owned by an application (respectively the Java Card RE) that can be "legally" invoked by another application or the Java Card RE (respectively an application). SIO methods and other container-managed objects' lifecycle methods are application entry point methods. Java Card RE entry point objects' methods are Java Card RE entry point methods.

entry point objects

see Java Card RE entry point object.

export file

a file produced by the Converter tool used during classic applet application development that represents the fields and methods of a package that can be imported by classes in other classic applet applications and classic libraries.

externally visible

in the Java Card platform, any classes, interfaces, their constructors, methods, and fields of an application that can be accessed from another application according to the Java programming language semantics, as defined by the *Java Language Specification*.

Externally visible items are represented in an export file. For a library package, externally visible items are represented in an export file. For an applet package, only those externally visible items that are part of a shareable interface are represented in an export file.

A Java Card CAP file may restrict the visibility of a package it contains. In this case, these packages are only visible to the other packages inside the CAP file and are not be accessible by packages in other CAP files. No export file is generated for the packages that have their visibility restricted to packages inside the same CAP file.

finalization

the process by which a Java virtual machine (JVM) allows an unreferenced object instance to release non-memory resources (for example, close and open files) prior to reclaiming the object's memory. Finalization is only performed on an object when that object is ready to be garbage collected (meaning, there are no references to the object).

Finalization is not supported by the Java Card virtual machine. The method `finalize()` is not called automatically by the Java Card virtual machine.

firewall

the mechanism that prevents unauthorized accesses to objects in one application group context from another application group context.

flash memory

a type of persistent mutable memory. It is more efficient in space and power than EPROM. Flash memory can be read bit by bit but can be updated only as a block. Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole.

framework

the set of classes that implement the API. This includes core and extension packages. Responsibilities include applet selection, sending APDU bytes, and managing atomicity.

garbage collection

the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.

global array

an array objects accessible from any context.

group context

protected object space associated with each `CAP` file and Java Card RE defining the boundaries of the firewall.

heap

a common pool of free memory in volatile and persistent spaces usable by a program for dynamic memory allocation, in which blocks of memory are used in an arbitrary order. The Java Card virtual machine's heap is not required to be garbage collected and objects allocated from the heap are not necessarily reclaimed.

installer

the on-card mechanism to download and install `CAP` files. The installer receives executable binary from the off-card installation program, writes the binary into the smart card memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card Runtime Environment.

installation program

the off-card mechanism that employs a card acceptance device (CAD) to transmit the executable binary in a `CAP` file to the installer running on the card.

instance variables

also known as non-static fields.

instantiation

in object-oriented programming, to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.

instruction

a statement that indicates an operation for the computer to perform and any data to be used in performing the operation. An instruction can be in machine language or a programming language.

internally visible

code items that are not externally visible. These items are not described in a package's export file and use private tokens to represent internal references. See externally visible

JAR file

an acronym for Java Archive file, which is a file format used for aggregating and compressing many files into one.

Java Card Platform Remote Method Invocation

a subset of the Java Platform Remote Method Invocation (RMI) system optionally supported by the Java Card RE. It provides a mechanism for a client application to invoke a method on a remote object of an applet on the card.

Java Card Runtime Environment (Java Card RE)

consists of the Java Card virtual machine, the application framework, 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.

Java Card RE context

the context of the Java Card RE has special system privileges so that it can perform operations that are denied to contexts of applications.

Java Card RE entry point object

an object owned by the Java Card RE context that contains entry point methods. These methods can be invoked from any context and allows applications to request Java Card RE system services. A Java Card RE entry point object can be either temporary or permanent:

temporary - references to temporary Java Card RE entry point objects cannot be stored in class variables, instance variables or array components. The Java Card RE detects and restricts attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. Examples of these objects are APDU objects and the APDU byte array.

permanent - references to permanent Java Card RE entry point objects can be stored and freely reused. Examples of these objects are Java Card RE-owned AID instances.

JDK software

an acronym for Java Development Kit. The JDK software provides the environment required for software development in the Java programming language. The JDK software is available for a variety of operating systems.

library CAP file

a CAP file that contains only library packages. See library package.

library package

a Java programming language package that does not contain any non-abstract classes that extend the class `javacard.framework.Applet`. See also applet package.

local variable

a data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and cannot be used outside the method.

logical channel

as seen at the card edge, works as a logical link to an applet application on the card. A logical channel establishes a communications session between a card applet and the terminal. Commands issued on a specific logical channel are forwarded to the active applet on that logical channel. For more information, see the *ISO/IEC 7816 Specification, Part 4*. (`http://www.iso.org`).

MAC

an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.

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.

method

a procedure or routine associated with one or more classes in object-oriented languages.

multiselectable applets

implements the `javacard.framework.MultiSelectable` interface. Multiselectable applets can be selected on multiple logical channels in the APDU application environment at the same time. They can also accept other applets belonging to the same applet application being selected simultaneously.

multiselected applet

an applet instance that is selected and, therefore, active on more than one logical channel in the APDU application environment simultaneously.

namespace

a set of names in which all names are unique.

native method

a method that is not implemented in the Java programming language, but in another language. The `CAP` file format does not support native methods to prevent from loading untrusted code.

nibble

four bits.

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.

object-oriented

a programming methodology based on the concept of an *object*, which is a data structure encapsulated with a set of routines, called *methods*, which operate on the data.

object

in object-oriented programming, unique instance of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

origin logical channel

the logical channel in the APDU application environment on which an APDU command is issued.

owning context

the application or Java Card RE context in which an object is instantiated or created.

owner context

see owning context.

package

a namespace within the Java programming language that can have classes and interfaces.

PCD

an acronym for Proximity Coupling Device. The PCD is a contactless card reader device.

persistent object

persistent objects and their values persist from one card session to the next, indefinitely. Objects are persistent when referred from another persistent object.

Persistent object values are typically updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized and deserialized, just that the objects are not lost when the card loses power.

PIX

see AID (application identifier).

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

functional and fully compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.

remote interface

an interface of an applet application, which extends, directly or indirectly, the interface `java.rmi.Remote`.

Each method declaration in the remote interface or its super-interfaces includes the exception `java.rmi.RemoteException` (or one of its superclasses) in its `throws` clause.

In a remote method declaration, if a remote object is declared as a return type, it is declared as the remote interface, not the implementation class of that interface.

In addition, Java Card RMI imposes additional constraints on the definition of remote methods of an applet application. See *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.1*.

remote methods

the methods of a remote interface of an applet application.

remote object

an object of an applet application whose remote methods can be invoked remotely from the off-card client. A remote object is described by one or more remote interfaces of an applet application.

RFU

acronym for Reserved for Future Use.

RID

see AID (application identifier).

RMI

an acronym for Remote Method Invocation. RMI is a mechanism for invoking instance methods on objects located on remote virtual machines (meaning, a virtual machine other than that of the invoker).

ROM (read-only memory)

memory used for storing the fixed program of the card. A smart card's ROM contains operating system routines as well as permanent data and user applications. No power is needed to hold data in this kind of memory. ROM cannot be written to after the card is manufactured. Writing a binary image to the ROM is called masking and occurs during the chip manufacturing process.

runtime environment

see Java Card Runtime Environment (Java Card RE).

service

a shareable interface object that a server application uses to provide a set of well-defined functionalities to its clients.

shareable interface

an interface that defines a set of shared methods. These interface methods can be invoked from an application in one context when the object implementing them is owned by an applet in another context.

shareable interface object (SIO)

an object that implements the shareable interface.

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.

SPI

an acronym for Service Provider Interface or sometimes for System Programming Interface. The SPI defines calling conventions by which a platform implementer may implement system services.

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.

The Java Card virtual machine can support only a single thread of execution. Java Card technology programs cannot use class Thread or any of the thread-related keywords in the Java programming language.

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.

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.

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).

verification

a process performed on an application or library executable that checks that the binary representation of the application or library is structurally correct and type safe.

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).

word

an abstract storage unit. A word is large enough to hold a value of type `byte`, `short`, `reference` or `returnAddress`. Two words are large enough to hold a value of `integer` type.