# Java Card 3 Platform

## Development Kit User Guide, Classic Edition

Version 3.0.5u3
E59602-03
April 2018

Java Card 3 Platform Development Kit User Guide, Classic Edition, Version 3.0.5u3

E59602-03

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents

## Preface

## Part I Setup, Samples and Tools

## 1 Introduction

## 2 Installation

## 3 Developing Classic Edition Applications

## 4 Running the Samples

## 13 Verifying CAP and Export Files

## 14 Using Cryptography Extensions

# Part II Programming With the Development Kit

## 15 Localizing With The Development Kit

## 16 Programming to the Java Card RMI Client-Side API

## 17 Working with APDU I/O

# List of Tables

# Preface

This document describes how to use the Java Card 3 Platform, Classic Edition Development Kit, version 3.0.5u3 to develop classic applets.

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

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

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

## Audience

This *Development Kit User's Guide* is written for developers who are creating classic applets using the *Application Programming Interface, Java Card Platform, Version 3.0.5, Classic Edition*, and also for developers who are considering creating a vendor-specific framework based on the Java Card specifications.

## Before You Read This Document

Before reading this guide, you should be familiar with the Java programming language and smart card technology.

You should also become familiar with the Java Card specifications, which are located at `https://docs.oracle.com/javacard`.

The Java Card binary is available for download 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:

- *Application Programming Interface, Java Card Platform, Version 3.0.5, Classic Edition*
- *Virtual Machine Specification, Java Card Platform, Version 3.0.5, Classic Edition*
- *Runtime Environment Specification, Java Card Platform, Version 3.0.5, Classic Edition*
- *Application Programming Notes, Java Card Platform, Version 3.0.5, Classic Edition*
- *Off-Card Verifier for the Java Card Platform White Paper*
- *Java Card Technology for Smart Cards by Zhiqun Chen (Prentice Hall, 2000)*
- *Java Card RMI Client Application Programming Interface* (see the Javadoc tool generated API specification at `JC_CLASSIC_HOME\docs\rmiclientlib`)
- *ISO 7816-4:2013 Specification*

## Documentation and Support

These web sites provide additional resources:

- Documentation `https://docs.oracle.com/javacard`
- 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
- Compatibility for Classic Applets
- Working With CAP Files
- Debugging Applications
- Packaging and Deploying Your Application
- Using the Reference Implementation
- Producing a Mask File from Java Card Assembly Files
- Building a Custom RI From Sources
- Verifying CAP and Export Files
- Using Cryptography Extensions

# 1 Introduction

The Java Card 3 Platform consists of two editions, the Classic Edition and the Connected Edition. This document and this development kit applies only to the Classic Edition.
The Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications. Applets that run on the Classic Edition are referred to as classic applets. The classic applets have the same capabilities as applets in previous versions of the development kit.

The Java Card development kit ships in binary-only bundles and in bundles with both binary and source versions of the kit. In addition, cryptography extensions are available in some bundles. All sections in this document pertain to all these types of bundles, except where noted. For the contents in the bundles, see the Release Notes.

This chapter contains the following sections:

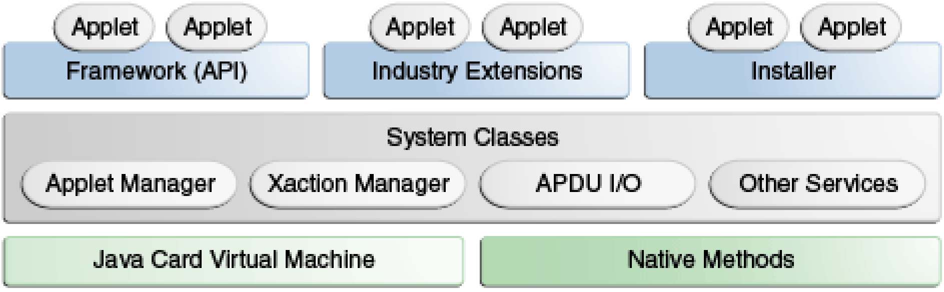
- Java Card 3 Platform Architecture
- Java Card TCK

## Java Card 3 Platform Architecture

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

The Classic Edition architecture illustrated is built on the classic Java Card VM, which is similar to the VM from previous releases of the Java Card development kit starting with version 2.2.2. Likewise, the classic APIs are similar to the APIs from previous releases.

**Figure 1-1 Classic Edition Architecture**



This development kit includes a Java Card RE that simulates a Java Card 3 Platform, Classic Edition as it would be implemented onto a smart card. This Java Card RE is the reference implementation (RI), and is invoked on the command line with `cref.bat`. The RI implements the ISO 7816-4:2013 specification, including support for up to twenty logical channels and the extended APDU extensions as defined in ISO 7816-3. For more information on the RI, see Using the Reference Implementation.

## Java Card TCK

The Java Card Technology Compatibility Kit (Java Card TCK) is a portable, configurable automated test suite for verifying the compliance of your implementation with the applicable Java Card specification. To be in compliance, your implementation must pass the Java Card TCK tests as described in the *Java Card Technology Compatibility Kit, Version 3.0.5u2, Classic Edition User's Guide*.

# 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.
Source and binary code development kits are available. Source code bundles allow you to change the development kit's reference implementation, whereas the binary bundles only allow you to use the reference implementation.

This chapter contains the following sections:

- Install and Setup the Development Kit
- Installed Files and Directories
- Setting Up the Eclipse IDE
- Uninstalling the Development Kit

## Install and Setup the Development Kit

This section describes how to install and set up the development kit. It includes procedures for performing the following tasks:

- Prerequisites to Installing the Development Kit
- Installing the Development Kit
- Confirming System Variables

### Before Installing the Development Kit

Before installing the development kit, be sure to install the following software:

- **Java Development Kit (JDK)** - The tools in this development kit were tested with JDK 7 and JDK 8 (32 bit and 64 bit versions). If you are planning to develop your own applications, you should use JDK 7. You can download and install the JDK release according to the instructions on the website:

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

- **GCC compiler** - Minimal GNU for Windows (MinGW), version 4.8.1 or later 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`.

- **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 from the following URL and install it according to the 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 1.9.4 was used to test the release. You can download and install Apache Ant from `http://ant.apache.org`.

## Installing the Development Kit

Follow these steps to install the development kit.

The Java Card binary development kit 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 `.msi` file to a directory of your choice.
   - `java_card_kit-classic-3_0_5-`*`ux`*`-win32-bin-`*`do|gl`*`-bNN-`*`dd_mmm_yyyy`*`.msi`
3. Run the downloaded `.msi` file from the directory.
   a. The Java Card Development Kit 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_CLASSIC_HOME`* throughout the documentation.

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

# Installed Files and Directories

The files and directories are installed under the root installation directory which you specify during installation. The root installation directory is referred to as *JC_CLASSIC_HOME* in this guide.

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

# Eclipse Java Card Plug-in

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

   **Oracle Java Card Tools for Eclipse IDE**
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_CLASSIC_HOME* environment variable, and
- Sample_Device, which contains the settings for cref (the simulator). See Using the Reference Implementation for more information about cref and its options.

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.

# Uninstalling the Development Kit

To uninstall the development kit, do the following:

1. Start Eclipse and remove the Java Card plug-in:
   a. From the Eclipse **Help** menu, select **Installation Details**.
   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** from the list of programs.
   c. Click **Uninstall** and then **Finish**.
3. Delete the *JC_CLASSIC_HOME* directory from your hard drive, if desired.

3

# Developing Classic Edition Applications

This chapter provides an introduction to developing Java Card applications. You should also refer to the *Application Programming Notes, Java Card Platform, Version 3.0.5, Classic Edition* for additional information not provided in this guide about creating applications for the Java Card 3 platform.
This chapter contains the following sections:

- Classic Applet Development Process
- Classic Development Kit Tools
- Using the Classic Tools

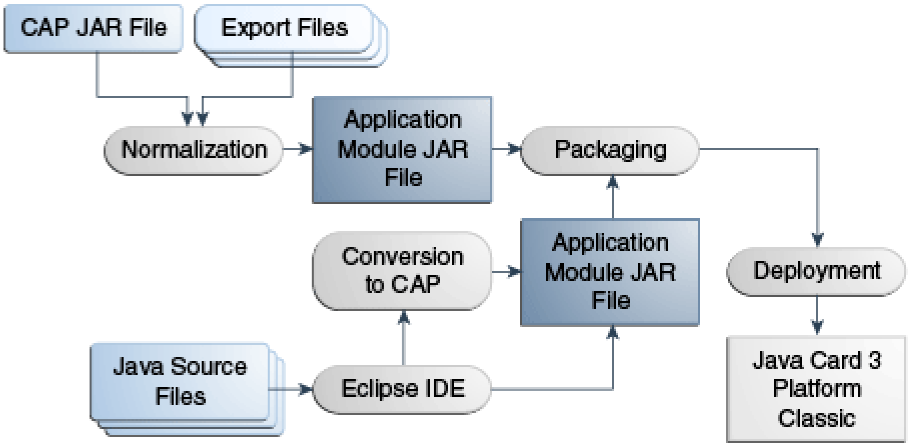
## Classic Applet Development

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. See Localizing With The Development Kit and other chapters in Part II for more information about various programming issues.
- **Debug** — Debug the applet. Use the Java Card debug proxy included in the development kit. See Debugging Applications.
- **Create CAP files** — When you have compiled your code to generate class files, you can use the tools provided with the development kit to create CAP files that can be downloaded to the simulator or a card. See Packaging and Deploying Your Application

The figure shows the deployment process for classic applet applications.

**Figure 3-1 Process for Classic Applet Development and Deployment**



If you have existing CAP files in Java Card platform 2.x format, you can use the normalization process to convert them to a version 3 CAP file format.

# Classic Development Kit Tools

The following tools are included in the development kit:

- **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
- **apdutool** - A client-side tool used to send 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.
- **capdump** - Creates an ASCII version of a CAP file. See Working With CAP Files.
- **capgen** - Generates a CAP file from a Java Card Assembly file. See Working With CAP Files.
- **classic_simulator** - Allows you to test your Java classes within the Eclipse IDE or from the command line. See Debugging Applications.
- **Converter** - Converts Java classes into a CAP file, a Java Card Assembly file, or an export file. See Converting and Exporting Java Class Files.
- **cref** - Runs the RI from the command line. See Using the Reference Implementation. There are three versions of `cref` to handle various communication protocols.
- **exp2text** - Enables you to view any export file in text format. See Converting and Exporting Java Class Files.
- **on-card installer** - The on-card installer resides on the smart card and downloads Java Card technology packages to a smart card. It can also delete packages and applets. See Packaging and Deploying Your Application.
- **maskgen** - Produces a mask file from a set of Java Card Assembly files. See Producing a Mask File from Java Card Assembly Files.

- **Normalizer** - Allows version 2.2.2 applets to be deployed on the Java Card 3 Platform. See Compatibility for Classic Applets.
- **off-card verifier** - Verifies the contents of a smart card using `verifycap`, `verifyexp`, and `verifyrev`. See Verifying CAP and Export Files.
- **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.
- **optional Ant tasks** - Additional, optional, and unsupported Ant tasks that can streamline development by combining the command line tools into useful groups of tasks. See Setting Up the Optional Ant Tasks.

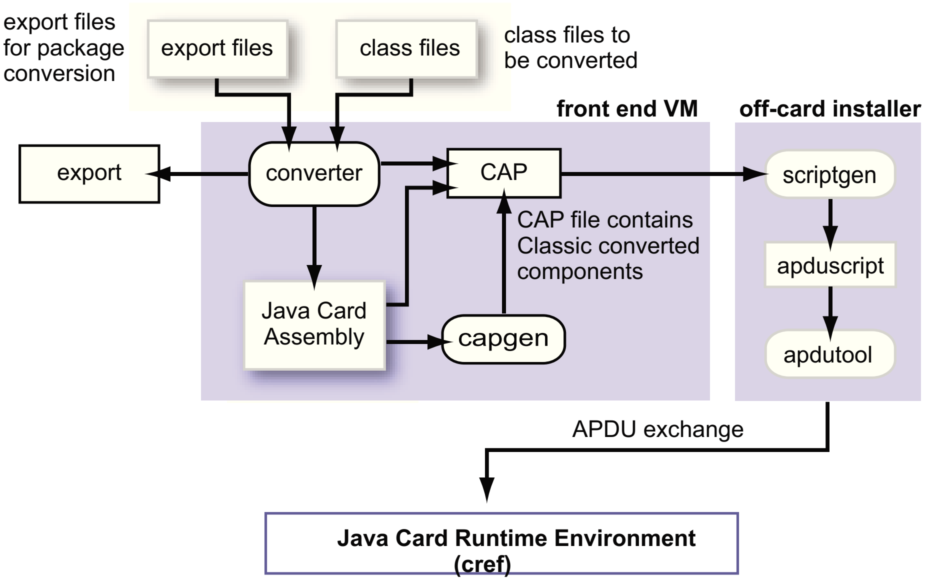
## Using the Classic Tools

Once you have written your applet, you convert it into a converted applet (CAP) file, package it, and send it through APDUs to the RI (`cref`) or, when the applet is complete, to a smart card containing a Java Card RE implementation.

The Converter tool converts a Java package into a CAP file or into a Java Card Assembly file. A CAP file is a binary representation of the converted package. A Java Card Assembly file is a text representation of a converted package that you can use to aid testing and debugging. A Java Card Assembly file can also be used as input to the `capgen` tool to create a CAP file.

CAP files are "packaged" by an off-card installer, `scriptgen`. It produces an APDU script file which is used by `apdutool`, to send APDUs to `cref` or to an off-card installer on the smart card.

**Figure 3-2 Java Card Platform Conversion**



Not shown in this diagram are the off-card verification tools, or the `capdump` tool, which produces a simple ASCII version of the CAP file to aid in debugging.

# 4 Running the Samples

A number of example programs are provided with the development kit.
Two directories containing samples are located under *JC_CLASSIC_HOME*\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 3 Platform 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** and **General** to import one or more projects at once, including both Java and Java Card projects. 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.

   See Using the Reference Implementation for more information about using `cref` and its command line options.

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

   `ant` *target*

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

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

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

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

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

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

# Running the classic_applets Samples

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

- HelloWorld Sample - A minimal applet utilizing the simplest source code and meta-files that demonstrates the base structure of a Java Card 3 platform 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 3 platform service framework of classes and interfaces that enable a Java Card technology-based applet to be designed as an aggregation of service components.
- 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 3 Platform, Classic Edition reference implementation.
- 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 intended solely for distribution inside the U.S.
- SignatureMessageRecovery Sample - Demonstrates message recovery. This sample is only included in bundles intended solely for distribution inside the U.S.

## 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

We will 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** and **General** 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. 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 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_CLASSIC_HOME*`\bin` directory.
   b. Start the RI by entering the following command at the command prompt:

      ```
      cref
      ```

      > **Note:**
      >
      > `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:
   a. Navigate to the *JC_CLASSIC_HOME*`\samples\classic_applets\HelloWorld\applet` directory.
   b. Enter the `ant all` command at the command prompt.

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

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

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

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

## Channels Sample

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

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

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

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

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

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

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

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

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_CLASSIC_HOME*`\bin` directory.

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

      ```
      cref
      ```

      > **Note:**
      >
      > `cref` command options are not required in this sample.

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

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

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

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

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

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

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

## Service Sample

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

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

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

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 `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:

   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 `Service`

   c. Click **Browse...**, select the Service 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_CLASSIC_HOME`*`\samples\classic_applets\Service\applet\apdu-scripts` directory and choose `cap-com.sun.jcclassic.samples.service.script`
      - From the same directory, select `service.scr`

   f. 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_CLASSIC_HOME*`\bin` directory.
   b. Start the RI by entering the following command at the command prompt:

      ```
      cref
      ```

      > **Note:**
      >
      > `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:
   a. Navigate to the JC_CLASSIC_HOME`\samples\classic_applets\Service\applet` directory.
   b. Enter the `ant all` command at the command prompt.

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

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

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

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

# Utility Sample

The `Utility` sample demonstrates how 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 **CREEF** 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_CLASSIC_HOME`*`\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_CLASSIC_HOME`\bin` directory.

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

   ```
   cref
   ```

   > **Note:**
   >
   > `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:
   1. Navigate to the JC_CLASSIC_HOME`\samples\classic_applets\Utility\applet` directory.
   2. 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:
   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. 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.

4. 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.

5. 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_CLASSIC_HOME`\bin` directory.
   b. Start the RI by entering the following command at the command prompt:

      ```
      cref
      ```

      > **Note:**
      >
      > `cref` command options are not required in this sample.

2. Open a second Command Prompt window and perform the following:
   a. Navigate to the JC_CLASSIC_HOME`\samples\classic_applets\Wallet\applet` directory.
   b. Enter the `ant all` command at the command prompt.

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

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

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

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

## ObjectDeletion Sample

The sample generates four 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. This sample has one script, od2.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

We will 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. In the Package Explorer, expand the apdu_scripts folder, and execute the following scripts, in order.

   - cap-com.sun.jcclassic.samples.odsample.libPackageC.script
   - cap-com.sun.jcclassic.samples.odsample.packageA.script

- `cap-com.sun.jcclassic.samples.odsample.packageB.script`

To run each script:

a. 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.

4. Execute these scripts, also in the `apdu_scripts` folder, starting the **Sample_Device** each time as described in the last step:

- `od1-1.scr`
- `od1-2.scr`
- `od1-3.scr`
- `od2.scr`
- `od2-2.scr`
- `od3.scr`
- `od3–2.cr`

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.

## 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_CLASSIC_HOME`\bin` directory.

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

```
cref -o e2p
```

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

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

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

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

3. In the `cref` Command Prompt window, stop the RI by using `ctrl` + `c`.

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

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

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

```
ant run1-1
```

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

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

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

6. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `od1-1.expected.out` file.
7. In the `cref` Command Prompt window, restart the RI by entering the following command:

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

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

   ```
   ant run1-2
   ```

   The `ant run1-2` command executes the `od1-2.scr` APDU script and creates an output file (`default.out`) in the `applet` directory. See Step 5 for the command line required to specify a custom output file name.
9. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `od1-2.expected.out` file.
10. In the `cref` Command Prompt window, restart the RI by entering the following command:

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

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

    ```
    ant run1-3
    ```

    The `ant run1-3` command executes the `od1-3.scr` APDU script and creates an output file (`default.out`) in the `applet` directory. See Step 5 for the command line required to specify a custom output file name.
12. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `od1-3.expected.out` file.
13. In the `cref` Command Prompt window, restart the RI by entering the following command:

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

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

    ```
    ant run2
    ```

    The `ant run2` command executes the `od2.scr` APDU script and creates an output file (`default.out`) in the `applet` directory. See Step 5 for the command line required to specify a custom output file name.
15. Verify that the contents of the output file in the `applet` directory are the same as the contents of the `od2.expected.out` file.
16. In the `cref` Command Prompt window, restart the RI by entering the following command:

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

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

    ```
    ant run2-2
    ```

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

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

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

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

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

    ```
    ant run3
    ```

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

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

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

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

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

    ```
    ant run3-2
    ```

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

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

## PhotoCard Sample

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

The `PhotoCard` sample consists of two parts: a card applet and a client program 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 an RI 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:

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

   b. Clear the **Output file with EEPROM data**.

   c. Select **Do not open APDU console**.
   d. 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:
    a. Right-click on **Java Application** and select **New**.
    b. Select the **Arguments** tab.
    c. 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:
   a. Navigate to the *JC_CLASSIC_HOME*`\bin` directory.
   b. Start the RI by using the following command at the command prompt:

      ```
      cref -o demoee
      ```

   Starting the RI with the `-o` option and *filename* causes the RI to save the EEPROM contents to a file named `demoee`. See Using the Reference Implementation for more information about using `cref` and its command line options.

2. Open a second Command Prompt window and perform the following:
   a. Navigate to the *JC_CLASSIC_HOME*`\samples\classic_applets\PhotoCard\applet` directory.
   b. Enter the following command at the command prompt:

      ```
      ant all
      ```

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

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

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

   Starting the RI with the `-z` and `-i` options and *filename* causes the RI 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:

   a. Navigate to the *JC_CLASSIC_HOME*`\samples\classic_applets\PhotoCard\client` directory.
   b. Enter the following command at the command prompt:

      ```
      ant all
      ```

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

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

> **Note:**
>
> Photo verification requires 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).

## 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. 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 **Program** in the left hand list of launch configuration types, and then click **New launch configuration** in the toolbar.
2. On the Create, manage, and run configurations dialog, enter the following text for these fields:
   - **Name:** `rmic`
   - **Location:** `${env_var:JAVA_HOME}\bin\rmic.exe`
   - **Working directory:** `${project_loc}`
   - **Arguments:** `–nowarn –v1.2 -classpath ".;${project_classpath}" -d .\stubs ${java_type_name}`

   Click **Apply** and **Close**.
3. 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.
4. 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**.
5. 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.

6. 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`
7. Stop the Sample_Device.
8. 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.
9. 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**.
10. 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.
11. 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:
   a. Navigate to the `*JC_CLASSIC_HOME*\bin` directory.
   b. Start the RI by typing the following command at the command prompt:

   ```
   cref -o demoee
   ```

   Starting the RI with the `-o` option and *filename* causes the RI to save the EEPROM contents to a file named `demoee`. See Using the Reference Implementation for more information about using `cref` and its command line options.
2. Open a second Command Prompt window and perform the following:
   a. Navigate to the *JC_CLASSIC_HOME*`\samples\classic_applets\RMIPurse\applet` directory.
   b. Enter the following command at the command prompt:

   ```
   ant all
   ```

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

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

   ```
   cref -i demoee
   ```

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

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

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

   b. Enter the following command at the command prompt:

      ```
      ant all
      ```

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

5. Verify that the contents of the rmidemo.actual.output file in the client directory are the same as the contents of the rmidemo.expected.output file in the 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. In the Package Explorer view, expand the StringHandlingApp project, right-click on com.sun.jcclassic.samples.stringapp package and select **Java Card** and **Package Settings**. Select the **ScriptGen** tab:
   a. Select **Suppress "PowerUp;" APDU command at the beginning of CAP script**
   b. Click **OK**
3. Repeat Step 2 for:
   - com.sun.jcclassic.samples.stringutilapp package in the same project, and
   - com.sun.jcclassic.samples.stringliblocal in the StringHandlingAppLibLocal project.

   You have modified these three packages. Next step is to build them again.
4. Rebuild the three modified projects: select **Project** and **Clean**, and from the Clean dialog, select the three projects that you just modified.

   Eclipse builds the projects again.
5. 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**.
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. Execute the scripts in the following order:
   - cap-com.sun.jcclassic.samples.stringlib
   - cap-com.sun.jcclassic.samples.stringliblocal
   - cap-com.sun.jcclassic.samples.stringapp
   - cap-com.sun.jcclassic.samples.stringutilapp
   - 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_CLASSIC_HOME*\bin directory, and start the RI by entering the following command at the command prompt:

   ```
   cref -o stringapp
   ```

2. Open a second Command Prompt window, navigate to the *JC_CLASSIC_HOME* `\samples\classic_applets\StringHandlingApp\lib` directory, and, at the command prompt, enter:

   ```
   ant all
   ```

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

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

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

   ```
   ant all
   ```

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

   ```
   cref -i stringapp
   ```

6. In the second Command Prompt window, navigate to the *JC_CLASSIC_HOME* `\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 available only in bundles intended solely for distribution inside the U.S.

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

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

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

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

The applet, `SecurePurseApplet`, is similar to the non-secure case, 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:

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

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

    ```
    cref -o demoee
    ```

    Starting the RI with the -o option causes the RI to save the EEPROM contents to a file named demoee. See Using the Reference Implementation for more information about using cref and its command line options.

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

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

    b. Enter the following command at the command prompt:

    ```
    ant all
    ```

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

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

    ```
    cref -i demoee
    ```

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

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

    b. Enter the following command at the command prompt:

    ```
    ant all
    ```

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

5. Verify that the contents of the securermidemo.expected.out 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 available only in bundles intended solely for distribution inside the U.S.

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

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:
   1. Select **Do not open APDU console**.
   2. Click **OK**.
3. Now create the first Run Configuration for this project. In the top menu, select **Run** and **Run Configurations...**
4. In the Run Configurations dialog:
   1. Right-click on **Java Card Project Run** and select **New**.
   2. In the **Name** field, enter `SignatureMessageRecovery_PartRec`
   3. Click **Browse...**, select the `SignatureMessageRecovery` project, and click **OK**.
   4. Select **Start simulator**.
   5. In the **Scripts to be executed on simulator** list box, add the following scripts:
      - `cap-com.sun.jcclassic.samples.signaturemessagerecovery.script` from `samples\classic_applets\SignatureMessageRecovery\applet\apdu-scripts`
      - `sigMsgPartRec.scr` from the same directory.
   6. Click **Apply** and **Close**
5. Create the second Run Configuration for this project. In the top menu, select **Run** and **Run Configurations...**
6. In the Run Configurations dialog:
   1. Right-click on **Java Card Project Run** and select **New**.
   2. In the **Name** field, enter `SignatureMessageRecovery_FullRec`
   3. Click **Browse...**, select the `SignatureMessageRecovery` project, and click **OK**.
   4. Select **Start simulator**.
   5. In the **Scripts to be executed on simulator** list box, add the following scripts:
      - `cap-com.sun.jcclassic.samples.signaturemessagerecovery.scri`

pt from `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**

7. 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.

8. 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_CLASSIC_HOME*`\bin` directory.

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

   ```
   cref
   ```

   > **Note:**
   >
   > `cref` command options are not required in this sample.

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

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

   b. Enter the following command at the command prompt:

   ```
   ant run1
   ```

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

3. Verify the contents of the `sigMsgPartRec.actual.output` file in the `applet` directory are the same as the contents of the `sigMsgPartRec.expected.output` file in the `SignatureMessageRecovery` directory.

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

   ```
   cref
   ```

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

   ```
   ant run2
   ```

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

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

# Running the reference_apps Samples

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

- `Biometry` Sample Application - Demonstrates the use of the biometric APIs of type PASSWORD.

  See Biometry Sample Application.

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

  This sample is only included in bundles intended solely for distribution inside the U.S. See JavaPurseCrypto Sample.

- `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 intended solely for distribution inside the U.S. 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. 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**.
3. 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**
4. 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**.
5. 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.

## Run the Biometry Sample from the Command Line

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

      ```
      cref -o e2p
      ```

   The RI saves the EEPROM contents to a file named `e2p`. See Using the Reference Implementation for more information about using `cref` and its command line options.

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

       ```
       ant all
       ```

       In this sample's `applet` directory, the `ant all` command executes the APDU script and installs the secure RMI application.
3. In the `cref` Command Prompt window, restart the RI by using the following command:

   ```
   cref -i e2p
   ```

   The RI 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_CLASSIC_HOME*`\samples\reference_apps\Biometry\Client\applet` directory.
    b. Enter the following command at the command prompt:

       ```
       ant all
       ```

       In this sample's `client` directory, the `ant all` command executes the APDU script.
5. Verify that the output displayed in the Command Prompt window is the same as the contents of the `biometry-client.expected.out` file.

## JavaPurse Sample Application

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

The `JavaPurse` applet demonstrates a simple electronic cash application. The applet is selected and initialized with various parameters such as the Purse ID, the expiration date of the card, the Master and User PINs, maximum balance, and maximum transaction. Transaction operations perform the actual debits and credits to the electronic purse. If a configured loyalty applet is assigned for the CAD performing the transaction, `JavaPurse` communicates with it to grant loyalty points. In this sample, `JavaLoyalty` is the provided loyalty applet.

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

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

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

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

## 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. 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**.
3. In the Edit Configuration window do the following:
   1. Enter `JavaPurse` in the **Name** field
   2. Select the **Start simulator** checkbox
   3. Add the following scripts to the listbox:
      - `cap-com.sun.jcclassic.samples.samplelibrary.script`
      - `cap-com.sun.jcclassic.samples.javaloyalty.script`
      - `cap-com.sun.jcclassic.samples.javapurse.script`
      - `jp.scr`
   4. Click **Apply** and **OK** to close the Edit Configuration window.
   5. Click **OK** to close the Properties window.
4. In Java Card View, double-click **Sample_Device**. In the Properties for Sample_Device dialog, select the **CREF** tab:
   1. Select **Do not open APDU console**.
   2. Click **OK**.
5. 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:
   1. Navigate to the *JC_CLASSIC_HOME*`\bin` directory.
   2. Start the RI by using the following command at the command prompt:

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

      ```
      ant all
      ```

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

ant -Dredirect.output=*outputfile_name target*

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

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

# JavaPurseCrypto Sample

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

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

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

As in the JavaPurse sample, the JavaLoyalty applet is a minimalistic loyalty applet that interacts with JavaPurseCrypto and demonstrates the use of shareable interfaces. See JavaPurse Sample Application 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. 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-com.sun.jcclassic.samples.samplelibrary.script`
   - `cap-com.sun.jcclassic.samples.javaloyalty.script`
   - `cap-com.sun.jcclassic.samples.javapursecrypto.script`
   - `jpcrypto.scr`

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

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

4. 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**.

5. 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_CLASSIC_HOME*`\bin` directory.

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

   ```
   cref
   ```

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

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

   b. Enter the following command at the command prompt:

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

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

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

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

## Transit Sample

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

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

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

Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the actual output files generated by running this sample 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_CLASSIC_HOME*`\bin` directory.
   b. Start the RI by using the following command at the command prompt:

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

      ```
      ant all
      ```
      In this sample's `applet` directory, the `ant all` command generates the APDU script and downloads the CAP file.
3. In the `cref` Command Prompt window, restart the RI by using the following command:

   ```
   cref -i transitCard -o transitCard
   ```
   Starting the RI with the `-i transitCard -o transitCard` options and filenames causes the RI to use the contents of the `transitCard` file to initialize the EEPROM and to save the EEPROM contents to a file named `transitCard`. See Using the Reference Implementation for more information about using `cref` and its command line options.
4. In the applet Command Prompt window, perform the following:
   a. Navigate to the *JC_CLASSIC_HOME*`\samples\reference_apps\Transit\Transit\client` directory.
   b. Enter the following command at the command prompt:

      ```
      ant run1
      ```
      In this sample's `client` directory, the `ant` run1 command compiles and builds the `client.jar` and generates the `actual_output1.txt` file.
5. Verify that the contents of the `actual_output1.txt` file are the same as the contents of the `TransitClient_1.expected.output` file.

   Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output1.txt` file 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 RI by using the following command:

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

```
ant run2
```

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

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

   Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output2.txt` file 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 RI by using the following command:

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

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

    ```
    ant run3
    ```

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

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

    Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output3.txt` file 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 RI by using the following command:

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

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

    ```
    ant run4
    ```

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

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

    Because the terminal uses random number generation for challenge/response and for generating session key, the contents of the `actual_output4.txt` file 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 preprocesses all of the Java class files that make up a package, and converts the package to a CAP file. The Converter also produces an export file.

Checks on the input classes include:

- Must be legal according to the Java Card Virtual Machine specification
- Must be Java SE 7 or earlier.
- All input class files are compatible with each other.
- Export files of imported packages are consistent with class files that were used for compiling the converting package.

The Converter generates the following output files:

- A CAP file, which is a JAR-format file which contains the executable binary representation of the classes in a Java package.
- Java Card Assembly file.
- Export file.

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

The Converter tool can only convert one package at a time. If you are converting more than one package with interdependencies, convert the packages in two passes. First, generate only the export files, then, after that, convert the required CAP or Java Card Assembly files.

If you have a source release, you may choose to convert packages that import other packages. If you are creating Java Card Assembly files to generate a mask, then the major and minor version number of the imported packages must agree with the version number of the package that imports them. See Version Numbers for Processed Packages for more information.

If you have a source release, you can localize locale-specific data associated with the Converter. Before you use the Converter tool, be sure to compile your Java code properly as described in Setting Java Compiler Options. For more information, see Localizing With The Development Kit.

For more information on the CAP file and its format, see the *Virtual Machine Specification, Java Card Platform, Version 3.0.5, Classic Edition*. The CAP file also contains a manifest file that provides human-readable information regarding the package that the CAP file represents. For more information on the manifest file and its contents, see Working With CAP Files.

For more information on the Java Card Assembly file, see Java Card Assembly Syntax Example and Producing a Mask File from Java Card Assembly Files. For more information on export files, see Using Export Files.

# Setting Java Compiler Options

To set Java compiler options:

1. Compile your class files with the Java SDK compiler's `-g` command line option.

The `-g` option causes the compiler to generate the `LocalVariableTable` attribute in the class file. The Converter uses this attribute to determine local variable types.

If you do not use the `-g` option, the Converter attempts to determine the variable types on its own. This is expensive in terms of processing and might not produce the most efficient code.You must also compile your class files with the `-g` option if you want to generate a debug component in the CAP file by using the Converter's `-debug` option.

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

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

# Running the Converter

To run the Converter:

1. Invoke the Converter by enter the following at the command line:

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

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

The Converter command line options described in Table 5-1 allow you to:

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

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

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

1. **Loads the package** - If the `exportmap` option is set, the converter loads the package from the export path (see Specifying an Export Map). It loads the class files of the Java package and creates a data structure to represent the package.
2. **Subset checking** - Checks for unsupported Java features in class files.
3. **Conversion** - Checks for consistency between the applet AIDs and the imported package AIDs.
4. **Reference Checking** - Checks that all references are valid, internal referenced items are defined in the package, 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 an export file, it does not check private APIs and byte code. Also included is a second round of subset checking that operations do not exceed the limitations set by the JCVM specification.
5. **Optimization** - Optimizes the bytecode.
6. **Generates output** - Builds and outputs the EXP export file and the JCA file, checks the package version in the export file of the current package against the package version specified in the command line. If the `-exportmap` option is used in the command line, the export file specified in the command line must represent the same version as that of the package. The converter does not support upgrading the export file version.

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

   *root_dir*\*package_dir*\\`javacard`

By default, the *root_dir* is the class root directory specified by the `-classdir` option. Users can specify a different *root_dir* by using the `-d` option.

**Table 5-1 Converter Command Line Arguments**

| Option | Description |
|---|---|
| `-help` | Prints help message. |
| *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. |
| *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.<br>**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` | Cannot be used with `-out` [*CAP*]. Indicates this package is for a mask, so restrictions on native methods are relaxed.<br>If you have a source release, you can specify this option to generate a mask out of this package using `maskgen`. |
| `-nobanner` | Suppresses all banner messages. |
| `-noverify` | Suppresses the verification of input and output files. For more information on file verification, see Verification of Input and Output Files . |
| `-nowarn` | Instructs the Converter not to report warning messages. |
| `-out` [*CAP*] [*EXP*] [*JCA*] | Cannot be used with `-mask`. Instructs the Converter to output the CAP file, and/or the export file, and/or the Java Card Assembly file. By default (if this option is not specified), the Converter outputs a CAP file and an export file. |
| `-v`, `-verbose` | Enables verbose output. Verbose output includes progress messages, such as "opening file", "closing file", and whether the package requires integer data type support. |
| `-V`, `-version` | Prints the Converter version string. |
| `-sign` | Specifies to sign the output CAP file |

**Table 5-1 (Cont.) Converter Command Line Arguments**

| Option | Description |
|---|---|
| `-keystore` *value* | Keystore to use in signing |
| `-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.<br>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:<br>*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.<br>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). |

# Using Delimiters with Command Line Options

To use delimiters with command line options:

1. 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_CLASSIC_HOME*`\api_export_files`, and current directories.

```
converter -exportpath ".\export files;.; JC_CLASSIC_HOME\api_export_files"

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

## Using a Command Configuration File

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

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:

   `-exportpath ".\export files;.;` *JC_CLASSIC_HOME*`\api_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.

# 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 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. 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.

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.

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

## Output File Naming Conventions

The name of the CAP file, export file, and the Java Card Assembly file must be the last portion of the package specification followed by the extensions`.cap`,`.exp`, and`.jca`, respectively.

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

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

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

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

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

## Verification of Input and Output Files

By default, the Converter invokes the Java Card technology-based off-card verifier ("Java Card off-card verifier") for every input EXP file and on the output CAP and EXP files.

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

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

## Creating a debug.msk Output File

To create a `debug.msk` output file:

1. Set the `-mask` and `-debug` options described in Table 5-1 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 (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.

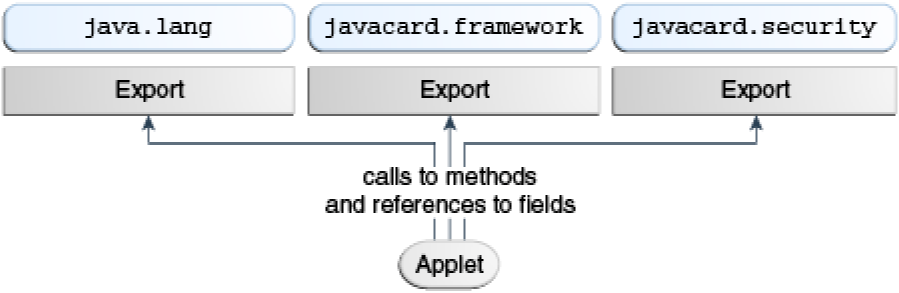
For more information on export files, see Verifying CAP and Export Files.

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 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 that the Converter 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 is desired:

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

  During package reimplementation, the API of the package (exportable classes, interfaces, fields and methods) must remain 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 *Virtual Machine Specification, Java Card Platform, Version 3.0.5, Classic Edition*).

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 5-1 when you run the Converter.

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

## Viewing an Export File as Text

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

To view an export file as text:

1. Enter the following command (see Table 5-2 for a description of the options):

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

**Table 5-2 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. |

If you have a source release, you can localize locale-specific data associated with the `exp2text` tool. For more information, see Localizing With The Development Kit.

# 6
# Compatibility for Classic Applets

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

This chapter contains the following sections:

- Generating Application Modules From Classic Applets

## Generating Application Modules From Classic Applets

Developers use the Normalizer to generate application modules for Java Card 3 Platform classic applets they are creating or from classic applets created for previous versions of the Java Card platform. These application modules contain CAP files and are downloadable on both the Java Card 3 platform Classic Edition and Connected Edition smart cards.

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

### Running the Normalizer

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

To run the Normalizer:

1. Enter the following command using the appropriate subcommands and options:

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

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

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

### normalize Subcommands

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

## normalize Subcommand Options

Table 6-1 identifies the `normalize` subcommand options and provides their descriptions.

**Table 6-1 normalize Subcommand Options**

| Option | Description |
|---|---|
| `-i` *file*<br>or<br>`--in` *file* | Specifies the input CAP file name. |
| `-p` *path*<br>or<br>`--exportpath` *path* | Specifies the path of the export files used by the tool. |
| `-f` *file*<br>or<br>`--exportfile` *file* | Specifies the export files used by the tool. |
| `-o` *directory*<br>or<br>`--out` *directory* | (Optional) This the default setting and does not have to be explicitly set. Specifies the output directory that contains the export file. |
| `-k`<br>or<br>`--keepall` | Specifies the directory to keep class files, proxy classes, and CAP components. The output format is as follows:<br>*directory*/*application classes*/`proxy`/*[proxy classes]*/`javacard`/ `*.cap` |

## normalize Subcommand Format

The following is the format of the `normalize` subcommand. Options in the subcommand are used in the sequence that are presented in Table 6-1. In this format example, an input file and an output directory are specified as options:

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

## normalize Subcommand Example

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

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

# copyright Subcommand

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

# help Subcommand

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

## Normalizer Summary Help

The following command displays summary help about the Normalizer:

```
normalizer.bat help
```

## normalize Subcommand Help

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

```
normalizer.bat help normalize
```

# 7
# 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 describes 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.

> **Note:**
>
> The `capgen` and `capdump` tools work only with CAP files generated with versions 2.2.2 and earlier of the Java Card development kit. This chapter contains a sample of the syntax used in CAP files as they apply to version 2.2.2 CAP files and related tools. The CAP file syntax was updated in the Development Kit version 3.*x*. This chapter deals primarily with the `capgen` and `capdump` tools and the earlier manifest file format that is still valid and supported with the 3.*x* Classic Edition.

This chapter contains the following sections:

- CAP File v2.2.2 Manifest File Syntax
- Generating a CAP File From a Java Card Assembly File
- Producing a Text Representation of a CAP File

## CAP File v2.2.2 Manifest File Syntax

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

The information in the manifest file is presented in name:value pairs. These name:value pairs are described in Table 7-1.

**Table 7-1 Name:Value Pairs in the MANIFEST.MF File**

| Name | Value |
|---|---|
| `Java-Card-CAP-Creation-Time` | Creation time of CAP file. For example:<br>Tue Jan 15 11:07:55 PST 2006<br>The format of the time stamp is operating system-dependent. |
| `Java-Card-Converter-Version` | The version of the converter tool. For example: `1.3`. |
| `Java-Card-Converter-Provider` | Provider of the converter tool. For example:<br>Oracle Corporation |
| `Java-Card-CAP-File-Version` | CAP file *major*.*minor* version. For example: 2.1. |
| `Java-Card-Package-Version` | The *major*.*minor* version of package. For example: 1.0 |
| `Java-Card-Package-AID` | AID for the package. For example:<br>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
```

# Generating a CAP File From a Java Card Assembly File

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

## Running capgen

To run `capgen`:

1. Enter the following on the command line (see Table 7-2 for a description of the options):

`capgen.bat` [*options*] *filename*

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

**Table 7-2 capgen Command Line Options**

| Option | Description |
|---|---|
| `-help` | Prints a help message. |
| `-nobanner` | Suppresses all banner messages. |

**Table 7-2 (Cont.) capgen Command Line Options**

| Option | Description |
|---|---|
| *filename* | Specifies the Java Card Assembly file. |
| `-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. |

# Producing a Text Representation of a CAP File

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

If you have a source release, you can localize locale-specific data associated with the capdump tool. For more information, see Localizing With The Development Kit.

## Running capdump

To run `capdump`:

1. 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_CLASSIC_HOME*`\bin` in order for the code to execute properly.

8

# 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 technology-enabled IDE.
This chapter contains the following sections:

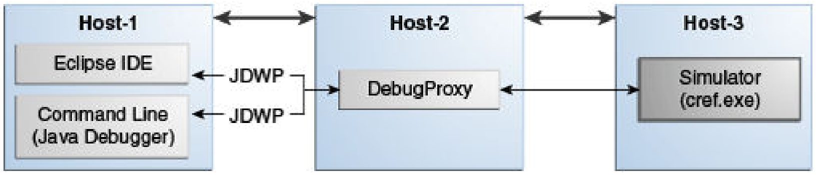
- Debugger Architecture
- Running the Debug Proxy from Eclipse
- Running the Debug Proxy From the Command Line

## Debugger Architecture

You can use `cref`, `debugproxy`, and an IDE to debug your project.

The pre-built 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 8-1 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, `debugproxy`, translates commands and responses between `cref` and the IDE into the appropriate protocol.

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

Ports used between the IDE and `debugproxy`, and `debugproxy` and `cref`, are distinguished by the names "listen port" and "remote port".

# Running the Debug Proxy from Eclipse

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, making sure that debugging information is generated when the project is built.
2. Create a Java Card debug configuration with your project settings:
   - You can specify scripts to be executed when the simulator starts.
   - Additionally, if you want to debug code from one or more libraries (packages without applet implementation), you can identify library cap file paths in the **Additional cap files for proxy** field. The cap path of the applet is automatically added; you do not need to identify it.
   - `cap-*` scripts for libraries should be executed before `cap-*` scripts for the applet if the EEPROM of `cref` is empty at the start of the debug session.
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. You can set breakpoints, and execute scripts.

Go to Debugging HelloWorld from Eclipse for detailed instructions for debugging the HelloWorld sample

## Debugging HelloWorld 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. Using the **File** menu, select **Import** and **General** 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. In the Package Explorer, expand the `HelloWorld` and src folder.

      You see the package `com.sun.jcclassic.samples.helloworld`.

   b. Right-click on `com.sun.jcclassic.samples.helloworld` and select **Java Card** and **Package Settings**.

   c. On the Java Card Package Settings page, select **Enable generation of debugging information**.
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…** options.

e. We need to add at least one script to be executed on the simulator before the debugger connects. Click **Add script...**. Browse to the `HelloWorld` project directory and select the `apdu_scripts\cap-com.sun.jcclassic.samples.helloworld.script` file. This script will install the applet without creating an applet instance.

f. 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:

   - Double-click the left most space on the source code line (the line number will be to the right).
   - Press **Ctrl + Shift + B** to toggle the breakpoint (the type of breakpoint will be selected automatically depending on the source code).
   - 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.

# 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_CLASSIC_HOME\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:
   1. Navigate to the *JC_CLASSIC_HOME*`\bin` directory.
   2. Start the RI 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:
   1. Navigate to the *JC_CLASSIC_HOME*`\samples\classic_applets\HelloWorld\applet` directory.
   2. 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.
   3. 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_CLASSIC_HOME*`\lib` directory and start the debug proxy:

   ```
   java.exe -jar jc-debug-proxy.jar -capPath JC_CLASSIC_HOME\samples
   \classic_applets\HelloWorld\applet\build\classes\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 localhost.
6. Now you can set a breakpoint and see it hit after a proper APDU is issued using the `apdutool`.

9

# 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 smart card. 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 package to a Java Card technology-compliant smart card, or during development, to the Java Card RE. Version 2.1 and later CAP files are supported.
- 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. For more information, see Using the On-card Installer for Deletion.
- 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`.

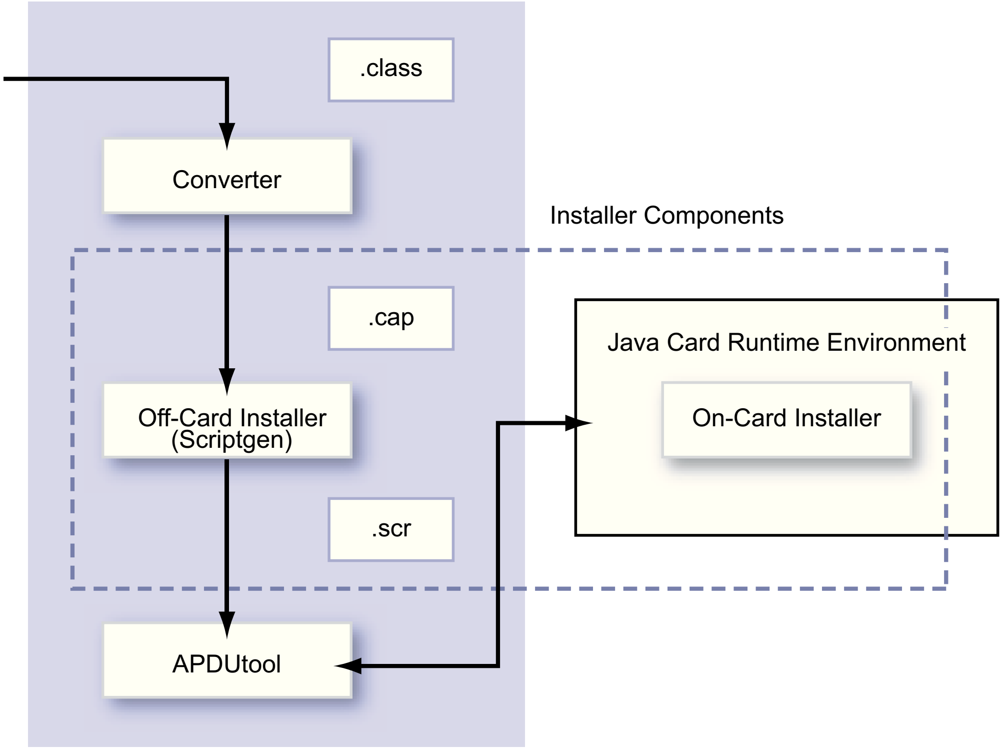
## Installer Components and Data Flow

Figure 9-1 illustrates the components of the installer and how they interact with other parts of Java Card technology. The dotted line encloses the installer components.

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

For more information about the installer, see the *Runtime Environment Specification, Java Card Platform, Version 3.0.5, Classic Edition*.

**Figure 9-1 Installer Components**



The data flow of the installation process is as follows:

1. `scriptgen` takes a version 2.1 or later CAP file produced by the Converter and generates a text file that contains a sequence of APDU commands.
2. This set of APDUs is read by `apdutool` and sent to the on-card installer.
3. The on-card installer processes the CAP file contents contained in the APDU commands, and sends a response APDU containing a status and, optionally, response data.

## Running scriptgen

The `scriptgen` tool converts a package contained in a CAP file into a script file. The script file contains a sequence of APDUs in ASCII format suitable for another tool, such as `apdutool`, to send to the CAD. The CAP file component order in the APDU script is identical to the order recommended by the *Virtual Machine Specification, Java Card Platform, Version 3.0.5, Classic Edition*. If you have a source release, you can localize data associated with the `scriptgen` tool.

1. 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_CLASSIC_HOME*`\bin` in order for the code to execute properly. Table 9-1 describes the options that can be used to invoke `scriptgen`.

**Table 9-1 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`). |
| `-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 RI 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. If you have a source release, you can localize messages from `apdutool`. For more information, see Localizing With The Development Kit.

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_CLASSIC_HOME*`\bin` in order for the code to execute properly.

To run the `apdutool`:

1. Enter the following command (see Table 9-2 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 9-2 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`<br>or<br>`-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:

1. 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:

1. 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 9-3 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 9-3 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. |

**Table 9-3 (Cont.) Supported APDU Script File Commands**

| Command | Description |
|---|---|
| `extended on;` | Turns extended APDU input mode on. |
| `extended off;` | Turns extended APDU input mode off. |
| `output off;` | Suppresses printing of the output. |
| `output on;` | Restores printing of the output. |
| `powerdown;` | Sends a `powerdown` command to the reader in the active interface. |
| `powerup;` | Sends a `powerup` command to the reader in the active interface. A `powerup` command must be sent to the reader prior to executing any APDU on the selected interface. |
| `select` *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 RI 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 RI prior to issuing this command.

**Table 9-4 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:

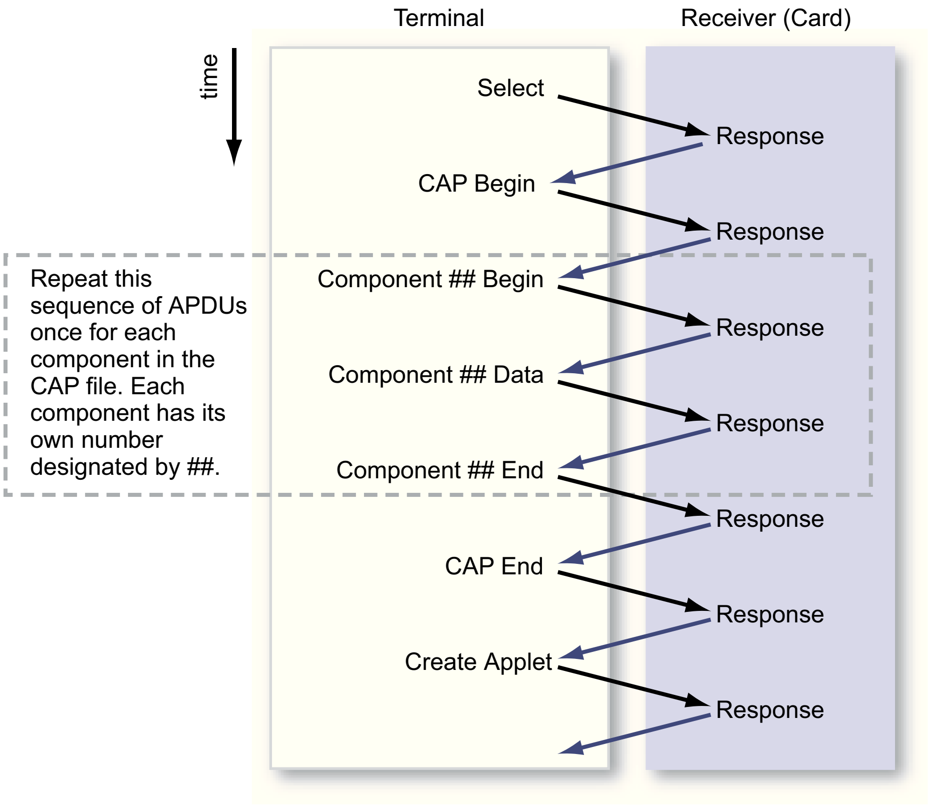
```
powerup;
// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08
    0x01 0x7F;
// create JavaPurse
0x80 0xB8 0x00 0x00 0x0b 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01
    0x04 0x01 0x00
0x7F;
powerdown;
```

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 9-2.

**Figure 9-2 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 9-5 specifies the field sequence in the `Select` APDU, which is used to invoke the on-card installer.

**Table 9-5 Select APDU Command**

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

### Response APDU Command

Table 9-6 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 9-6 Response APDU Command**

| Data | Response |
|---|---|
| [optional response data] | SW1SW2 |

## CAP Begin

Table 9-7 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 9-7 CAP Begin APDU Command**

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

## CAP End

Table 9-8 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 9-8 CAP End APDU Command**

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

## Component ## Begin

Table 9-9 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 9-9 Component ## Begin APDU Command**

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

## Component ## End

Table 9-10 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 9-10 Component ## End APDU Command**

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

## Component ## Data

Table 9-11 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 9-11 Component ## Data APDU Command**

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

## Create Applet

Table 9-12 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 9-12 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 9-13 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 9-13 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 9-14.

**Table 9-14 APDU Responses to Installation Requests**

| Response Code | Description |
|---|---|
| `0x6402` | Invalid CAP file magic number.<br>• **Cause:** An incorrect magic number was specified in the CAP file.<br>• **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.<br>• **Cause:** An invalid CAP file minor number was specified in the CAP file.<br>• **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.<br>• **Cause:** An invalid CAP file major number was specified in the CAP file.<br>• **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.<br>• **Cause:** An attempt was made to download a CAP file that requires integer support into a CREF that does not support integers.<br>• **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.<br>• **Cause:** A duplicate package AID was detected in CREF.<br>• **Solution:** Choose a new AID for the package to be installed. |
| `0x640d` | Duplicate Applet AID found.<br>• **Cause:** A duplicate Applet AID was detected in CREF.<br>• **Solution:** Choose a new AID for the applet to be installed. |
| `0x640f` | Installation aborted.<br>• **Cause:** Installation was aborted by an outside command.<br>• **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.<br>• **Cause:** A non-recoverable error previously occurred.<br>• **Solution:** Scan the `apdutool` output for previous APDU responses indicating an error. Restart the CAP installation. |
| `0x6422` | CAP file component out of order.<br>• **Cause:** Installer unable to proceed because it did not receive a component that is a prerequisite to process the current component.<br>• **Solution:** Check the script file contents for the correct component ordering. |
| `0x6424` | Exception occurred.<br>• **Cause:** General purpose error in the installer or applet code.<br>• **Solution:** Check your applet code for errors. |

**Table 9-14 (Cont.) APDU Responses to Installation Requests**

| Response Code | Description |
|---|---|
| `0x6425` | Install APDU command out of order.<br>• **Cause:** Installer APDU commands were received out of order.<br>• **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.<br>• **Cause:** An incorrect component tag number was detected during download.<br>• **Solution:** Refer to Chapter 6 in the *Java Virtual Machine Specification* for the correct tag number. |
| `0x6436` | Invalid install instruction.<br>• **Cause:** An invalid Installer APDU command was received.<br>• **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.<br>• **Cause:** Package installation failed because the number of packages that can be stored on the card has been exceeded.<br>• **Solution:** Remove some packages from the CREF. |
| `0x6438` | Imported package not found.<br>• **Cause:** A package that is required by the current package was not found.<br>• **Solution:** Download the required package first. |
| `0x643a` | On-card applet package max exceeded.<br>• **Cause:** Installation of an applet package failed because the number of applet packages that can be stored on the card has been exceeded.<br>• **Solution:** Remove some applet packages from the CREF. |
| `0x6442` | Maximum allowable package methods exceeded.<br>• **Cause:** The limit of 128 package methods on the card has been exceeded.<br>• **Solution:** Modify the package to support fewer methods. |
| `0x6443` | Applet not found for installation.<br>• **Cause:** An attempt was made to create an applet instance, but the applet code was not installed on the card.<br>• **Solution:** Verify that the applet package has been downloaded to the card. |
| `0x6444` | Applet creation failed.<br>• **Cause:** A general purpose error to indicate that an unsuccessful attempt was made to create the applet.<br>• **Solution:** Verify availability of resources on the card, check the applet's `install` method, and so on. |
| `0x644f` | Package name is too long.<br>• **Cause:** The package name exceeds the length specified in the *Java Virtual Machine Specification*.<br>• **Solution:** Replace the name and rebuild. |

**Table 9-14 (Cont.) APDU Responses to Installation Requests**

| Response Code | Description |
|---|---|
| `0x6445` | Maximum allowable applet instances exceeded.<br>• **Cause:** Creation of the applet instance failed because the number of applet instances that can be stored on the card has been exceeded.<br>• **Solution:** Remove some applet instances from the CREF. |
| `0x6446` | Memory allocation failed.<br>• **Cause:** The amount of memory available on the card has been exceeded.<br>• **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.<br>• **Cause:** A class that is required by the current class was not found.<br>• **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 reference implementation provides the ability to delete package and applet instances from the card's memory. Once the on-card installer is selected, it can receive deletion requests from the terminal in the form of ADPU commands. Requests to delete an applet or package cannot be sent from an applet on the card. For more information on package and applet deletion, see the *Runtime Environment Specification, Java Card Platform, Version 3.0.5, Classic Edition*.

## 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 Package

b. Delete Package 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 Packages and Applets

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

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

### Delete Package

In this request, the Data field contains the size of the package AID and the AID of the package to be deleted. Table 9-15 shows the format of the Delete Package request and the expected response.

**Table 9-15 Delete Package 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 Package Request:
 0x81 0xC0 0x00 0x00 0x08 0x07 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x7F;
```

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

### Delete Package and Applets

This request is similar to the Delete Package command. In this case the package and applets are removed simultaneously. The data field contains the size of the package AID and the AID of the package to be deleted. Table 9-16 shows the format of the Delete Packages and Applets request and the expected response.

**Table 9-16 Delete Package 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 package and applets deletion request on channel 1 would be:

```
//Delete Package And Applets request
0x81 0xC2 0x00 0x00 0x08 0x07 0xa0 0x00 0x00 0x00 0x62 0x12 0x34 0x7F;
```

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

## Delete Applets

In this request, the "#" symbol in the `P1` byte indicates the number of applets to be deleted, which can have a maximum value of eight. The `Lc` field contains the size of the data field. Data field contains a list of AID size and AID pairs. Table 9-17 shows the format of the Delete Applet request and the expected response.

**Table 9-17 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 9-18.

**Table 9-18 APDU Responses to Deletion Requests**

| Response Code | Description |
|---|---|
| `0x6a86` | Invalid value for `P1` or `P2` parameter.<br>• **Cause:** Value for `P1` is less than 1 or greater than 8.<br>• **Solution:** Ensure that the value for `P1` is between 1 and 8. |
| `0x6443` | Applet not found for deletion.<br>• **Cause:** The applet with the specified AID does not exist.<br>• **Solution:** Check and correct the AID. |
| `0x644b` | Package not found.<br>• **Cause:** The package with the specified AID does not exist.<br>• **Solution:** Check and correct the AID. |

**Table 9-18 (Cont.) APDU Responses to Deletion Requests**

| Response Code | Description |
|---|---|
| `0x644c` | Dependencies on package.<br>• **Cause:** Package has other packages dependent on it, or there are some object instances of classes belonging to this package residing in memory.<br>• **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.<br>• **Cause:** One or more applet instances of this package are present<br>• **Solution:** Remove the applets first and then try package deletion, or try the package and applet deletion combination command. |
| `0x644e` | Package is ROM package.<br>• **Cause:** An attempt was made to delete a package in ROM.<br>• **Solution:** There is no solution to this problem since packages in ROM cannot be deleted. |
| `0x6448` | Dependencies on applet.<br>• **Cause:** Other applets are using objects owned by this applet.<br>• **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.<br>• **Cause:** There is not enough memory available for the intermediate structures required by applet deletion.<br>• **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.<br>• **Cause:** An attempt was made to delete an applet which is currently active on one of the logical channels.<br>• **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.<br>• **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.<br>• **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 9-19.

**Table 9-19 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 packages to be downloaded is 32, including up to 16 applet packages.
- The maximum number of applet instances to be created is 16.
- The maximum length of data in the installer APDU commands is 128.
- No on-card CAP file verification is supported.
- All subsequent APDU commands enclosed in a `CAP Begin`, `CAP End` APDU pair continue to fail after an error occurs.
- The maximum number of applets that can be deleted using one command is eight.

10

# Using the Reference Implementation

This chapter describes how to run the Reference Implementation (RI). The RI provides a Java Card runtime environment (RE) executable for the Microsoft Windows platform, Java Card RE packages, and an installer applet.
This chapter contains the following sections:

## Overview of Using the Reference Implementation

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

The Java Card RI supports the following:

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

In this version of the development kit, the RI is available as a 32-bit implementation that supports the ability to go beyond the 64KB memory access. The RI supports the T=1 and T=CL in dual concurrent interface mode. This development kit also supports T=0 and T=1, both in single interface mode. Table 10-1 lists the Java Card RE executables by protocol.

**Table 10-1 Protocols Supported by RE Executables**

| RE Executable | Supported Protocol |
|---|---|
| `cref_t0.exe` | T=0 protocol (single port), as defined in ISO 7816. |
| `cref_t1.exe` | T=1 protocol (single port), as defined in ISO 7816. |

**Table 10-1 (Cont.) Protocols Supported by RE Executables**

| RE Executable | Supported Protocol |
|---|---|
| `cref_tdual.exe` | T=1 and T=CL (each on separate port). Uses T=1 for the contact protocol, and the T=CL for the contactless version, as defined in ISO 14473. This is the default executable for `cref.bat`. |

**Note:**

The descriptions, command-line syntax, and options in this chapter for running `cref_tdual.exe` also apply to `cref_t0.exe` and `cref_t1.exe`. The difference is solely in the supported protocols.

The binary versions of the REs available in this development kit are provided in the *JC_CLASSIC_HOME*`\bin` directory as the executables `cref_t0.exe`, `cref_t1.exe`, and `cref_tdual.exe`. Each binary corresponds to the supported protocols described in Table 10-1. The development kit also includes the `cref.bat` file whose default is to run the RI with T=1/T=CL, but it can also run the RI using the other supported protocols according to arguments shown in Table 10-2.

**Table 10-2 Case Sensitive Command Line Options for cref.bat**

| Option | Description |
|---|---|
| `[-t0 | -t1 | -tdual]` | Specifies the RE version to run according to the desired protocol (see Table 10-1). Defaults to `-tdual` to call `cref_tdual.exe` if not specified. |
| `-b` | Dumps a bytecode histogram at the end of the execution. |
| `-e` | Displays the program counter and stack when an exception occurs. |
| `-h`, `-help` | Prints a help screen. |
| `-i` *input-file-name* | Specifies a file to initialize EEPROM. There can be no spaces in the file name argument. |
| `-n` | Performs a trace display of the native methods that are invoked. |
| `-nobanner` | Suppresses the printing of a program banner. |
| `-nomeminfo` | Suppresses the printing of memory statistics when execution starts. |
| `-o` *output-filename* | Saves the EEPROM contents to the named file. |
| `-e2pfile` *file-name* | Specifies the EEPROM file. If the file exists it is read to initialize the EEPROM image. All `e2pfile` images are written to this file. This option is the same as combining `-i` and `-o` into one. |
| `-p` *port-number* | Connects to a TCP/IP port using the specified *port-number*. |

**Table 10-2 (Cont.) Case Sensitive Command Line Options for cref.bat**

| Option | Description |
|---|---|
| `-contactedport` *port-number* | Connects to a TCP/IP port using the specified *port-number*. |
| `-s` | Suppresses output. Does not create any output unless followed by other flag options. |
| `-t` | Performs a line-by-line trace display of the mask's execution. |
| `-version` | Prints only the program's version number. Do not execute. |
| `-z` | Prints the resource consumption statistics. |

## Running the RI

Invoke `cref` using `cref.bat` from *JC_CLASSIC_HOME*`\bin`

To run the RI:

1. Enter the following command (options are described in Table 10-2):

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

## Installer Mask

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

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

## Obtaining Resource Consumption Statistics

Use the RI `-z` command line option to print memory usage at startup and at shutdown. Although memory usage statistics vary among Java Card RE implementations, this give you general idea of the amount of memory needed to install and execute an applet.

> **Note:**
>
> The Java Card API provides programmatic mechanisms for determining resource usage. For more information, see the `javacard.framework.JCSystem.getAvailableMemory()` method in the *Application Programming Interface, Java Card Platform, Version 3.0.5, Classic Edition*.

## Getting Resource Statistics With the PhotoCard Sample

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

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

To get resource statistics with the photocard sample:

1. Enter the following command to run the `PhotoCard` sample program:

   ```
   JC_CLASSIC_HOME\bin> cref_tdual -z -o e2p
   ```

   Example 10-1 shows the output from running the `PhotoCard` sample program from the `bin` directory with the `-z` option.

**Example 10-1    PhotoCard Sample Showing Resource Statistic Output**

```
Java Card 3.0.5 C Reference Implementation Simulator
32-bit Address Space implementation - with cryptography support
T=1 / T=CL Dual interface APDU protocol (ISO 7816-4)
Copyright (c) 1998, 2015, Oracle and/or its affiliates. All rights reserved.

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

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

```
    Clear on Deselect, channel space 3 : 0 bytes
    Clear on Deselect, channel space 4 : 0 bytes
    Clear on Deselect, channel space 5 : 0 bytes
    Clear on Deselect, channel space 6 : 0 bytes
    Clear on Deselect, channel space 7 : 0 bytes
    Clear on Reset: 0 bytes

C:\JCDK3.0.5\bin>cref_tdual -z -i e2p
Java Card 3.0.5 C Reference Implementation Simulator
32-bit Address Space implementation - with cryptography support
T=1 / T=CL Dual interface APDU protocol (ISO 7816-3)
Copyright (c) 1998, 2015, Oracle and/or its affiliates. All rights reserved.

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

        ROM Mask size =                 0xcec9 =        52937 bytes
        Highest ROM address in mask =   0xeec8 =        61128 bytes
        Space available in ROM =        0x1137 =        4407 bytes
EEPROM (0xffe0 bytes) restored from file "e2p"
Using a pre-initialized Mask
        0 bytecodes executed.
        Stack size: 00384 (0x0180) bytes,          00000 (0x0000) maximum used
        EEPROM use: 09558 (0x2556) bytes consumed, 55946 (0xda8a) available
Transaction buffer: 00000 (0x0000) bytes consumed, 03560 (0x0de8) available
Clear-On-Reset RAM: 00198 (0x00c6) bytes consumed, 00378 (0x017a) available
Clear-On-Dsel. RAM: 00025 (0x0019) bytes consumed, 00231 (0x00e7) available
CREF was powered down.
  4624835 bytecodes executed.
        Stack size: 00384 (0x0180) bytes,          00250 (0x00fa) maximum used
        EEPROM use: 56993 (0xdea1) bytes consumed, 08511 (0x213f) available
Transaction buffer: 00000 (0x0000) bytes consumed, 03560 (0x0de8) available
Clear-On-Reset RAM: 00198 (0x00c6) bytes consumed, 00378 (0x017a) available
Clear-On-Dsel. RAM: 00125 (0x007d) bytes consumed, 00131 (0x0083) available
Temporary memory usage:
    Java stack: 0 bytes
    Clear on Deselect, channel space 0 : 0 bytes
    Clear on Deselect, channel space 1 : 0 bytes
    Clear on Deselect, channel space 2 : 0 bytes
    Clear on Deselect, channel space 3 : 0 bytes
    Clear on Deselect, channel space 4 : 0 bytes
    Clear on Deselect, channel space 5 : 0 bytes
    Clear on Deselect, channel space 6 : 0 bytes
    Clear on Deselect, channel space 7 : 0 bytes
    Clear on Reset: 0 bytes
```

## RI Limits

- The maximum number of remote references that can be returned during one card session is 8.
- The maximum number of remote objects that can be exported simultaneously is 16.
- The maximum number of parameters of type array that you can use in remote methods is 8.

- The maximum number of Java Card API packages that the Java Card RI can support is 32.
- The maximum number of library packages that a Java Card system can support is 32.
- The maximum number of applets that a Java Card system can support is 16.

# Input and Output

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

Use `apdutool` to read script files and send APDUs through a socket to the RI. See apdutool Examples for details. Note that you can have the Java Card RI running on one workstation and run `apdutool` on another workstation.

# Working With EEPROM Image Files

You can save the state of EEPROM contents, then load it in a later invocation of the RI. To do this, you must specify an EEPROM image or "store" file to save the EEPROM contents when running `cref`. The `-i` and `-o` flags are used to manipulate the EEPROM image files at the `cref` command line:

- The `-i` flag, followed by a file name, specifies the initial EEPROM image file that initialize the EEPROM portion of the virtual machine before Java Card virtual machine bytecode execution begins.
- The `-o` flag, followed by a file name, saves the updated EEPROM portion of the virtual machine to the named file, overwriting any existing file of the same name.

The `-i` and `-o` flags do not conflict with the performance of other option flags.

The commit of EEPROM memory changes during the execution of the Java Card RI is not affected by the `-o` flag. Neither standard nor error output is written to the output file named with the `-o` option.

This section includes the following topics:

- Input EEPROM Image File
- Output EEPROM Image File
- Same Input and Output EEPROM Image File
- Different Input and Output EEPROM Image Files

## Input EEPROM Image File

The following command example shows how you can use the `-i` flag to initialize EEPROM from an input EEPROM image file.

To initialize EEPROM from an input EEPROM image file:

1. Enter the following command:

   ```
   cref -i e2save
   ```

The RI attempts to initialize simulated EEPROM from the EEPROM image file named `e2save`. No output file is created.

## Output EEPROM Image File

The following command example shows how you can use the -o flag to writes EEPROM data to an output EEPROM image file.

To write EEPROM data to an output EEPROM image file:

1. Enter the following command:

   ```
   cref -o e2save
   ```

The Java Card RI writes EEPROM data in this example to the file e2save. The file is created if it does not currently exist. Any existing EEPROM image file named e2save is overwritten.

## Same Input and Output EEPROM Image File

The following example shows how you can use the -o and -i flag to initialize simulated EEPROM from the EEPROM image file, and during processing, save the contents of EEPROM to the same image file, overwriting the contents.

To initialize EEPROM from and write EEPROM data to the same input and output EEPROM image file:

1. Enter the following command:

   ```
   cref -i e2save -o e2save
   ```

The Java Card RI attempts to initialize simulated EEPROM from the EEPROM image file named e2save, and during processing, saves the contents of EEPROM to e2save, overwriting the contents. This behavior is much like a real Java Card technology-compliant smart card in that the contents of EEPROM are persistent.

## Different Input and Output EEPROM Image Files

The following example shows how you can use the -o and -i flag to initialize simulated EEPROM from an EEPROM image file, and during processing, write EEPROM updates to a different EEPROM image file.

1. Enter the following command:

   ```
   cref -i e2save_in -o e2save_out
   ```

The RI attempts to initialize simulated EEPROM from e2save_in, and during processing, writes EEPROM updates to the file named e2save_out. The output file is created if it does not exist. Using different names for input and output EEPROM image files eliminates much potential confusion.

This command line can be executed multiple times with the same results.

11

# Producing a Mask File from Java Card Assembly and CAP Files

This chapter describes how to use the maskgen tool to create a mask from Java Card Assembly and CAP files. The maskgen tool is not available or of use outside of a source release bundle, so you can disregard this chapter if you do not have a source release of the development kit. If you have a source release, you can localize locale-specific data associated with the maskgen tool, see Localizing With The Development Kit.

This chapter contains the following sections:

- Overview of Producing a Mask File from Java Card
- Running maskgen

## Overview of Producing a Mask File from Java Card

The maskgen tool produces a mask file from a set of Java Card Assembly and CAP files produced by the Converter. The format of the output mask file is targeted to a specific platform. The plug-ins that produce each different maskgen output format are called generators. The supported generators are cref, which supports the Java Card RE, and size, which reports size statistics for the mask. Other generators that are not supported in this release include jref, which supports the Java programming language Java Card RE, and a51, which supports the Keil A51 assembly language interpreter. Java Card Assembly Syntax Example provides additional information about the contents of a Java Card Assembly file.

## Running maskgen

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

To run maskgen:

1. Enter the following command (options are described in Table 11-1):

   ```
   maskgen [options] generator filename [filename ...]
   ```

**Table 11-1 Command Line Arguments for the maskgen Tool**

| Argument | Description |
|---|---|
| -help | Prints a help message. |

**Table 11-1 (Cont.) Command Line Arguments for the maskgen Tool**

| Argument | Description |
|---|---|
| *generator* | Specifies the generator, the plug-in that formats the output for a specific target platform. The generators are:<br>• `a51` - Output for the Keil A51 assembly language interpreter (not supported for this release).<br>• `cref` - Output for the `cref` interpreter.<br>• `jref` - Output for the Java programming language Java Card RE interpreter (not supported for this release).<br>• `size` - Outputs mask size statistics.<br>In this release, the only supported generator is `cref`. |
| *filename* `[`*filename*`...]` | Any number of Java Card Assembly and CAP files can be input to `maskgen` as a whitespace-separated list.<br>You can also create a text file containing a list of Java Card Assembly and CAP file names for a new mask, and prepend an "@" character to the name of this text file as an argument to `maskgen`. |
| `-c` *filename* | Specifies a configuration file, which contains generator-specific settings. For example, the following line maps a native Java Card API method to a native label:`javacard/framework/JCSystem/beginTransaction()V=beginTransaction_NM`<br>`cref_mask.cfg` is an example of a maskgen configuration file. |
| `-debuginfo` | Generates debug information for the generated mask. This option is available only with the `jref` generator. |
| `-nobanner` | Suppresses all banner messages. |
| `-o` *filename* | Specifies the file name output from `maskgen`. If the output file is not specified, output defaults to `a.out`. |
| `-version` | Prints the version number of `maskgen`, then exits. |

# Order of Packages on the Command Line

The Java Card Assembly files that can be listed on the command line can belong to API packages, the installer package, or the user's library and applet packages. CAP files for the user's library or applet packages may also be provided on the command line. The Java Card Assembly files that belong to API packages must be listed first on the command line, followed by the Java Card Assembly or CAP files belonging to any applets.

If you include the installer package's Java Card Assembly file on the command line, it must be listed after all of the assembly files belonging to API packages and before the assembly or CAP files of any other applet packages. If a Java Card CAP file is listed on the command line, the export files needed to verify the CAP file by the verifier must also be listed on the command line.

For example:

```
maskgen -nobanner cref API_package_1.jca .... API_package_n.jca
    installer_package.jca applet_package_1.jca ...
    applet_package_n.jca
    applet_package_a.CAP applet_package_a.exp lang.exp framework.exp
```

# Version Numbers for Processed Packages

The packages that you specify to generate a mask can import other packages. These imported packages must share the same major and minor version number as the specified packages.

For example, presume that you are using `Package A`, version 1.1 to create a mask, and that `Package A` imports `Package B`, version 1.1. Then you must ensure that `Package B`, version 1.1 is listed in the `import` component of the `Package A .jca` file.

# maskgen Example

This example uses a text file (`args.txt`) to pass command line arguments to `maskgen`:

```
maskgen -o mask.c cref @args.txt
```

where the contents of the file `args.txt` is:

```
first.jca second.jca third.jca
```

This is equivalent to the command line:

```
maskgen -o mask.c cref first.jca second.jca third.jca
```

This command produces an output file `mask.c` that is compiled with a C compiler to produce `mask.o`, which is linked with the Java Card RE interpreter. Refer to Using the Reference Implementation for more information about this target platform.

# 12 Building a Custom RI From Sources

This chapter describes how to build a custom Java Card RI from sources and only applies if you have the source release. The `src` folder contains all the files that are only in the source release, including the code for the virtual machine (VM) and all tools.

This chapter contains the following sections:

- Overview of Building a Custom RI from Sources
- Steps for Building a Custom RI

## Overview of Building a Custom RI from Sources

You can modify the RI by adding or modifying its code. The RI consists of `.java` and C source files. The core VM is written in the C programming language and the rest of the API and supported implementation is written in the Java programming language.

You can modify or add to these files and build a customized Java Card 3 platform RI according to specific requirements. You might want to build a custom RI for the following reasons:

- Add additional classes or packages if a proprietary API or other implementation classes are used.
- Fine tune the existing sources.
- Update the tools to work with a target platform.
- Mask an application into `cref`.

## Steps for Building a Custom RI

Before building a custom RI, the software listed in Prerequisites to Installing the Development Kit must be installed on the system. This section describes the basic step you must follow when creating a custom RI.

To build a custom RI:

1. Convert your packages using the Converter tool and generate Java Card Assembly files.

   > **Note:**
   >
   > Converting and Exporting Java Class Files provides a detailed description of using the Converter tool.

   For example, the output Java Card Assembly (`.jca`) files could be located at:

   ```
   .\api_export_files\com\sun\javacard\installer\javacard\installer.jca
   ```

or

*JC_CLASSIC_HOME*`\samples\classic_applets\HelloWorld\applet\build\classes\com\sun\jcclassic\samples\helloworld\javacard\helloworld.jca`

2. Add the location for the Java Card Assembly files for your new packages in these files:

   - `src\vm_16.in`
   - `src\vm_32.in`

   This step is required for new packages. If you are modifying existing packages, this step is not required.

3. Build the `cref` executable.

   > **Note:**
   >
   > Building the 32-Bit Custom RI, and Building the 16-Bit Custom RI provide detailed procedures for building the `cref` executable.

   The new `cref` is built with the new packages masked in. Masked applications can be instantiated without requiring download after the runtime environment starts up.

4. Test the custom RI.

   > **Note:**
   >
   > Testing the 32-Bit Custom RI and Building the 16-Bit Custom RI provide detailed procedures for testing different custom RIs.

## Building the 32-Bit Custom RI

The following procedure builds the 32-bit version of the RI. To build the 16-bit version see Building the 16-Bit Custom RI.

To build the 32-bit custom RI:

1. Edit the files or add more files.
2. Update the tools source code, if required.
3. From the command line, navigate to the `src` folder and run the `ant all` command.

   The `ant all` command creates the `JC_CLASSIC_HOME\custom_build` folder with a `bin` and `lib` folder under it.

   The `custom_build\bin` directory contains the new `cref` and all of the other `.bat` files for the tools.

   The `custom_build\lib` folder contains the `.jar` files and configuration files.

## Testing the 32-Bit Custom RI

To test the 32-bit custom RI:

1. Run the new `cref` file stored in *JC_CLASSIC_HOME*`\custom_build\bin` noting the expected console output in Example 12-1 and its specific reference in the content to the `32-bit Address Space`.

   ```
   JC_CLASSIC_HOME\custom_build\bin\cref_tdual.exe [options]
   ```

   See Using the Reference Implementation for a description of the available options for `cref`.

   Files created as a result of running or building the custom RI are stored in the *JC_CLASSIC_HOME*`\custom_build\bin` and *JC_CLASSIC_HOME*`\custom_build\lib` directories. These directories are created the first time the custom RI is built and are over-written every time the custom RI is built.

2. Run `apdutool` in a separate window to send in your apdu scripts to `cref`.

   ```
   apdutool -nobanner -noatr filename.scr > filename.scr.cref.out
   ```

   For information on `apdutool`, see Running apdutool. If the run is successful, the `apdutool.log`, *filename*`.scr.cref.out`, is identical to the file *filename*`.scr.expected.out`.

**Example 12-1 Expected Console Output When Building 32-Bit Custom RI**

```
Java Card 3.0.5 C Reference Implementation Simulator
32-bit Address Space implementation - with cryptography support
T=1 / T=CL Dual interface APDU protocol (ISO 7816-4)
Copyright (c) 1998, 2015, Oracle and/or its affiliates. All rights reserved.

Memory configuration -
        Type    Base    Size    Max Addr
        RAM     0x0     0xc78   0xc77
        ROM     0x2000  0xc800  0xe7ff
        E2P     0x10020 0xffe0  0x1ffff

        ROM Mask size =                 0x8b3e =        35646 bytes
        Highest ROM address in mask =   0xab3d =        43837 bytes
        Space available in ROM =        0x3cc2 =        15554 bytes
Mask has now been initialized for use
```

## Building the 16-Bit Custom RI

By default the procedure described in Building the 32-Bit Custom RI builds the 32-bit `cref`. To build `cref` with 16-bit support you must modify those steps.

**Note:**

Building the 16-bit version creates only one `cref_t0.exe` file with no `t1` or `tdual` version, so the resulting `cref.bat` file executes only `cref_t0.exe`.

To build the 16-bit custom RI:

1. Before starting, clean any previous builds by running `ant clean`.
2. When executing `ant all`, set this property: `for.bit=16`, with the following modified command:

   ```
   ant -Dfor.bit=16 all
   ```

3. When testing the 16-bit build, execute `cref.bat` from the `custom_build/bin` directory and watch for the expected output depicted in Example 12-2 noting its specific reference in the content to the `16-bit Address Space`.

### Example 12-2 Expected Console Output When Building 16-Bit Custom RI

```
Java Card 3.0.3 C Reference Implementation Simulator
16-bit Address Space implementation - with cryptography support
T=0 Extended APDU protocol (ISO 7816-3)
Copyright (c) 2010 Oracle Corporation All rights reserved.
Use is subject to license terms.

Memory configuration
        Type    Base    Size    Max Addr
        RAM     0x0     0x600   0x5ff
        ROM     0x600   0xbc00  0xc1ff
        E2P     0xc200  0x3be0  0xfddf

        ROM Mask size =                 0x9fb0 =        40880 bytes
        Highest ROM address in mask =   0xa5af =        42415 bytes
        Space available in ROM =        0x1c50 =        7248 bytes
Mask has now been initialized for use
```

# 13
# 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 smart card and the set of export files used to construct those CAP files, the Java Card technology-enabled off-card verifier provides the means to assert that the content of the smart card has been verified.
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

If you have a source release, you can localize data associated with the off-card verifier. Localizing With The Development Kit contains additional information about off-card verification.

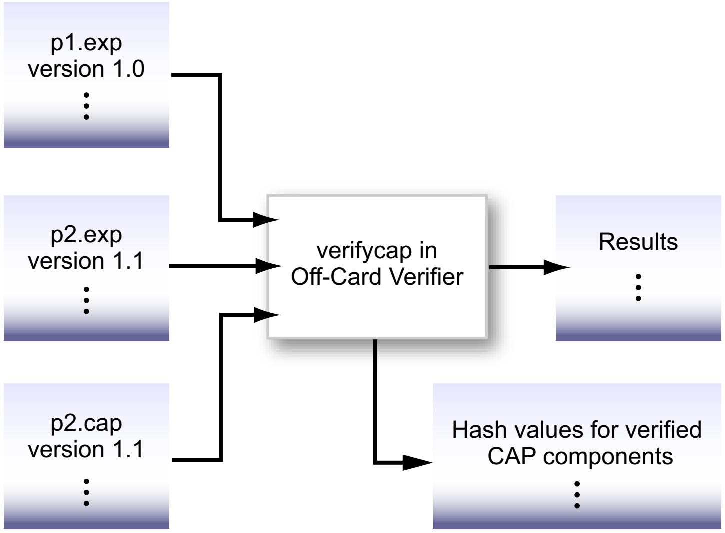
## Verifying CAP Files

The `verifycap` tool is used to verify a CAP file within the context of package's export file (if any) and the export files of imported packages. This verification confirms whether a CAP file is internally consistent, as defined in the *Virtual Machine Specification, Java Card Platform, Version 3.0.5, Classic Edition*, 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 13-1. 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 13-1    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_CLASSIC_HOME*`\bin` in order for the code to execute properly.

To run `verifycap`:

1. Enter the following command (Table 13-1 describes the available options):

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

**Table 13-1    verifycap Command Line Arguments**

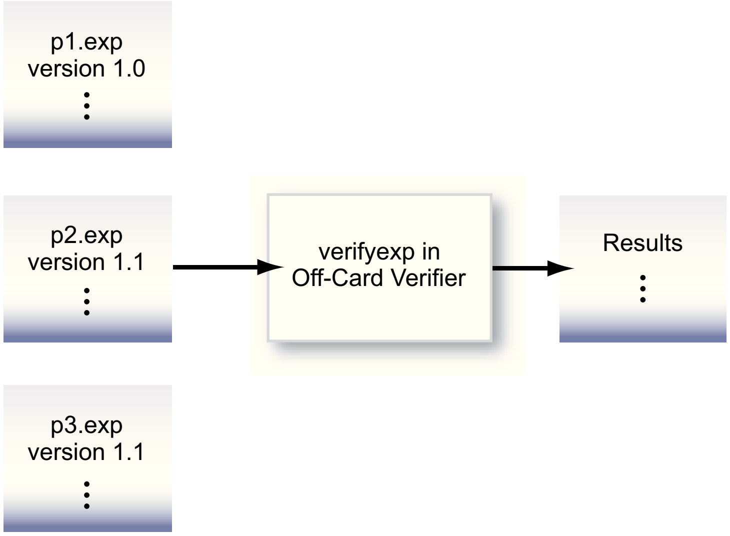
| Argument | Description |
|---|---|
| *export-files* | A list of export files of the packages that this CAP file uses. |
| *CAP-files* | Name of the CAP file to be verified. |
| `-digest` *`digest-algorithm-name`* | Specifies the digest algorithm to use for computing hash values for required CAP components. 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 *Virtual Machine Specification, Java Card Platform, Version 3.0.5, Classic Edition*. This scenario is illustrated in Figure 13-2.

**Figure 13-2 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_CLASSIC_HOME*`\bin` for the code to execute properly.

To run `verifyexp`:

1. Enter the following command (Table 13-2 describes the available options):

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

**Table 13-2 verifyexp Command Line Argument**

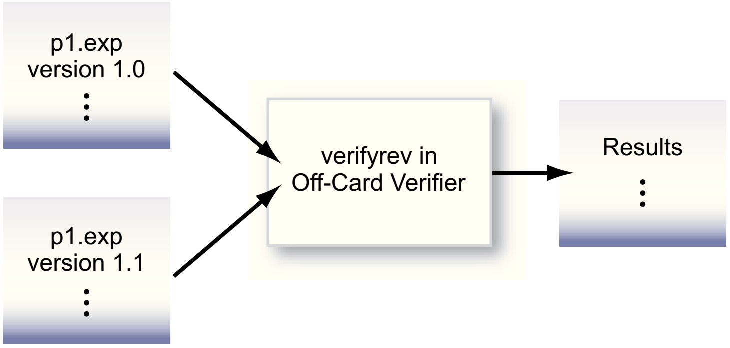
| Argument | Description |
|---|---|
| <*export file*> | Fully qualified path and name of the export file. |

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 13-3. 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 *Virtual Machine Specification, Java Card Platform, Version 3.0.5, Classic Edition*, have been followed.

**Figure 13-3 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_CLASSIC_HOME*`\bin` in order for the code to execute properly.

To run `verifyrev`:

1. 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 exception is the `-package` option which is available for `verifycap` only.

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

**Table 13-3 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. |
| `-verbose` | Enables verbose mode. |
| `-version` | Prints version number and exit. |
| `-C` *command-options-file*<br>or<br>`--commandoptionsfile` *command-options-file* | Optional. Specifies a file containing command-line options. |

# 14 Using Cryptography Extensions

This chapter describes how to use the basic security and cryptography classes, which do not appear in all Java Card development kits.

This chapter contains the following sections:

## Overview of Using Cryptography Extensions

Security and cryptography classes are supported by the RI (`cref`). The support for security and cryptography enables you to:

- Generate message digests using the `SHA1` algorithm
- Generate cryptographic keys on Java Card technology-compliant smart cards for use in the `ECC` and `RSA` algorithms
- Set cryptographic keys on Java Card technology-compliant smart cards for use in the `AES`, `DES`, `3DES`, `ECC`, and `RSA` algorithms
- Encrypt and decrypt data with the keys using the `AES`, `DES`, `3DES`, and `RSA` algorithms
- Generate signatures using the `AES`, `DES`, `3DES`, `ECC`, or `SHA` and `RSA` algorithms
- Generate sequences of random bytes
- Generate checksums
- Use part of a message as padding in a signature block

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

For more information on the `SHA1`, `DES`, `3DES`, and `RSA` encryption schemes, see:

- For `SHA1`—"*Secure Hash Standard*", FIPS Publication 180-1: `http://www.itl.nist.gov`
- For `DES`—"*Data Encryption Standard (DES)*", FIPS Publication 46-2 and "*DES Modes of Operation*", FIPS Publication 81: `http://www.itl.nist.gov`
- For `RSA`—"*RSAES-OAEP (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

# Supported Cryptography Classes

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

- `javacardx.crypto.AEADCipher`
- `javacardx.crypto.Cipher`
- `javacard.security.Checksum`
- `javacard.security.InitializedMessageDigest`
- `javacard.security.KeyAgreement`
- `javacard.security.KeyBuilder`
- `javacard.security.KeyPair`
- `javacard.security.MessageDigest`
- `javacard.security.RandomData`
- `javacard.security.Signature`
- `javacard.security.SignatureMessageRecovery`

> **Note:**
>
> The implementation of `RandomData` in the RI is not suitable for porting.

Table 14-1 lists the cryptography algorithms that are implemented for the RI.

**Table 14-1 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.<br>• `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. |

**Table 14-1 (Cont.) Algorithms Implemented by the Cryptography Classes**

| Class | Algorithm |
|---|---|
| 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.<br>• ALG_RSA_PKCS1—provides a cipher using RSA. Input data is padded according to the PKCS#1 (v1.5) scheme.<br>• ALG_AES_BLOCK_128_CBC_NOPAD—provides a cipher using AES with block size 128 in CBC mode and does not pad input data.<br>• 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].<br>• 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:<br>• 128-bit AES<br>• 64-bit DES<br>• 112-, 128-, 160-, 192-bit ECC<br>• 128-bit DES3<br>• 512-bit RSA |
| KeyPair | The algorithms define the key lengths for:<br>• 112-, 128-, 160-, 192-bit ECC<br>• 512-bit RSA |
| MessageDigest | Message digest algorithm SHA1 and SHA256 |
| 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.<br>• ALG_RSA_SHA_PKCS1—encrypts the 20 byte SHA1 digest using RSA. The digest is padded according to the PKCS#1 (v1.5) scheme.<br>• ALG_AES_MAC_128_NOPAD—generates a 16-byte MAC using AES with blocksize 128 in CBC mode and does not pad input data.<br>• ALG_ECDSA_SHA—signs/verifies the 20-byte SHA digest using ECDSA.<br>• ALG_AES_CMAC_128 |
| SignatureMessageRecovery | • ALG_RSA_SHA_ISO9796_MR—This algorithm uses the first part of the input message as padding bytes during signing. During verification, these message bytes (recoverable message) can be recovered to reconstruct the message. |

# Instantiating the Classes

Implementations of the cryptography classes extend the corresponding base class with implementations of their abstract methods. All data allocation associated with the implementation instance is performed when the instance is constructed. This is done to ensure that any lack of required resources can be flagged when the applet is installed.

Each cryptography class, except `KeyPair`, has a `getInstance` method which takes the desired algorithm as one of its parameters. The method returns an instance of the class in the context of the calling applet. Instead of using a `getInstance` method, `KeyPair` takes the desired algorithm as a parameter in its constructor.

If you request an algorithm that is not listed in Table 14-1 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's guide provides solutions for various programming issues. It contains the following chapters:

- Localizing With The Development Kit
- Programming to the Java Card RMI Client-Side API
- Working with APDU I/O
- Programming for the Large Address Space

# 15
# Localizing With The Development Kit

This chapter describes the support for localization that the development kit provides. This chapter is useful only if you have a source release of the development kit. This chapter contains the following sections:

- Overview of Localizing with the Development Kit
- Localization Support for Java Utilities
- Localization Support for cref

## Overview of Localizing with the Development Kit

Items that can be localized in the development kit include the Java language-based tools, `cref`, and the Java language-based Java Card RMI sample applications and client framework. The Java language-based programs and the C language-based programs use different localization mechanisms.

## Localization Support for Java Utilities

This section describes the mechanisms used to localize the following programs and tools:

- RMI sample programs
- RMI client framework
- `scriptgen`
- `apdutool`
- `converter`
- normalizer
- `maskgen`
- `capdump`
- `exp2text`
- off-card verifier

These Java utilities and programs can be localized in a similar fashion. Each uses the Java language resource bundle mechanism. This mechanism allows the user to customize locale-sensitive data for a new locale without rebuilding the application. Refer to the Java SE platform `java.util.ResourceBundle` class for more information regarding resource bundles.

The development kit also provides localization support for Java Card RMI sample applications and client framework. Localizing the client framework and the sample applications can be done in the same way as the Java Card technology-based utilities.

Since none of the Java Card platform reference implementation utilities or programs require a graphical user interface (GUI) and are not dependent on user input, the majority of the locale-specific data consists of static strings. Localization consists of customizing these strings for the intended locale. Locale-sensitive strings are grouped into `.properties` files (for example, `MessagesBundle.properties`). Localizing an application entails creating a new version of the properties file that contains the translated strings.

## Localizing a Java Program

To localize a Java program:

1. Create a new version of the property file which contains the set of strings customized for the intended locale.
2. Rename the property file with the appropriate locale identifier appended to the file name (for example, the French version of the `MessagesBundle.properties` file would be `MessagesBundle_fr.properties`).
3. Include the location of the property file in the CLASSPATH for the `cref` utility.

   When the Java utility is executed in an environment with the same locale as the properties file, the strings contained in that properties file are used for output.

# Localization Support for cref

Localizing `cref` consists of translating the set of static output strings used by `cref`. Unlike the Java language utilities, `cref` must be rebuilt for a new locale. This requires the source bundle of this development kit.

All of the locale-sensitive strings used by `cref` are stored in a single C header file, `src\vmshare\h\cref_locale.h`. To localize `cref`, customize the strings in this file and then rebuild `cref`.

16

# Programming to the Java Card RMI Client-Side API

This chapter describes how to use the Java Card RMI client-side API. A Java Card RMI client application runs on a Card Acceptance Device (CAD) terminal that supports a Java SE or Java ME platform.
This chapter contains the following sections:

- Overview of Programming to the Java card RMI Client Side
- Remote Stub Object
- Java Card RMI Client-Side API

## 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 RI 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_CLASSIC_HOME*`\lib\tools.jar`.

The reference implementation of the Java Card RMI client-side API is based on APDU I/O for its card access mechanisms. For more information on APDU I/O, see Working with 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 *Application Programming Notes, Java Card Platform, Version 3.0.5, Classic Edition*.

> **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 reference implementation 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 *Runtime Environment Specification, Java Card Platform, Version 3.0.5, Classic Edition*. Any object references must contain class names.

- **class JCCardProxyFactory**–The `JCCardProxyFactory` class is similar to `JCCardObjectFactory`, but processes references containing lists of names. `JCCardProxyFactory` uses the JDK 1.4.+ proxy mechanism to generate proxies dynamically.
- **class JCRemoteRefImpl**–An implementation of interface `java.rmi.server.RemoteRef`. These remote references can work with stubs generated by the RMIC compiler with the `-v1.2` option.

  The main method is: `public Object invoke(Remote remote, Method method, Object[] params, long unused) throws IOException, RemoteException, Exception`

  This method prepares the outgoing APDU, passes it to `CardAccessor`, and then uses `CardObjectFactory` to parse the returned APDU and instantiate the returned object or throw an exception.

## Package clientlib

This package includes an interface and a class.

- **interface CardAccessor**–An interface defining methods to exchange APDUs with a card and to close connection to a card.
- **class ApduIOCardAccessor**–A simple implementation of the CardAccessor interface that passes the APDUs to a card or a card simulator using the ApduIO library. This class takes parameters to start the ApduIO from the file `jcclient.properties`, which must be included in CLASSPATH.

# 17
# 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

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_CLASSIC_HOME*`\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 RI 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 RI is implemented in such a way that once the client establishes connection to a port, the next command must be `powerUp` on that port. For example, the following sequence is valid:

1. Connect on "contacted" port.
2. Send `powerUp` to it.
3. Exchange some APDUs.
4. Connect on "contactless" port.
5. Send `powerUp` to it.
6. Exchange more APDUs

However, the following sequence is not valid:

1. Connect on "contacted" port.
2. Connect on "contactless" port.

3. Send `powerUp` to any port.

To 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 RI) 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:

1. The following code prints both C-APDU and R-APDU in the `apdutool` output format.

   ```
   System.out.println(apdu)
   ```

18

# 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.
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 RI is the large address space. Allowing your applications to take advantage of the large address capabilities of the Classic Edition RI requires careful planning and programming. Some size limitations still exist within the reference implementation. The way that you structure large applications and applications that manage large amounts of data determines how the large address space can be exploited.

## Programming Large Applications and Libraries

The key to writing large applications for the Java Card 3 Platform, Classic Edition is to divide the code into individual package units. The most important limitation on a package is the 64KB limitation on the maximum component size. This is especially true for the Method component: if the size of an application's Method component exceeds 64KB, then the Java Card converter does not 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. Keep in mind that there are important differences between library packages and applet packages:

- In a library package, all public fields are available to other packages for linking.
- In an applet package, only interactions through a shareable interface are allowed by the firewall.

Therefore, you should not place sensitive or exclusive-use code in a library package. It should be placed in an applet package, instead.

## Handling a Package as a Separate Code Space

Several applications and API functionality can be installed in the smart card simultaneously by handling each package as a separate code space. This technique 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 reference implementation had to save downloaded applications or user data in valuable persistent memory space. Sometimes, the amount of memory space required was insufficient for some applications. However, the memory access schemes introduced with version 2.2.2 allow applications to store large amounts of information, while still conforming to the Java Card specification.

The Java Card specification does not impose any requirements on object location or total object heap space used on the card. It specifies only that each object must be accessible by using a 16-bit reference. It also imposes some limitations on the amount of information an individual object is capable of storing, by using the number of fields or the count of array elements. Because of this loose association, it is possible for any given implementation to control how an object's information is stored, and how much data these objects can collectively hold.

The Java Card 3 Platform, Classic Edition reference implementation, 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 reference implementation 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_CLASSIC_HOME*`\samples\classic_applets\PhotoCard\applet\src\com\sun\jcclassic\samples\photocard`

The collection of arrays (more than two arrays would be required in this case) can easily hold far more than 64KB of data. Storing this amount of information should not be a problem, provided that enough mutable persistent memory is configured in the RE.

# Part III

# 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 an annotated Java Card platform assembly ("Java Card Assembly") file output from the Converter. The comments in this file are intended to help you understand the syntax of the Java Card Assembly language, and to act as a guide for debugging 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_CLASSIC_HOME*`\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 {
    }
}
```

# 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_CLASSIC_HOME\lib\jctasks.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_CLASSIC_HOME\lib\jctasks.jar` to a directory that serves as your Ant tasks home directory.
3. Add the `jctasks.jar` file to your classpath or put it into the *Ant-Home-Path*`\lib` directory to be automatically be picked up when Ant is run.

   Where:

   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="capgen"
  classname="com.sun.javacard.ant.tasks.CapgenTask" />
<taskdef name="maskgen"
  classname="com.sun.javacard.ant.tasks.MaskgenTask" />
<taskdef name="deploycap"
  classname="com.sun.javacard.ant.tasks.DeployCapTask" />
<taskdef name="exp2text"
  classname="com.sun.javacard.ant.tasks.Exp2TextTask" />
<taskdef name="convert"
  classname="com.sun.javacard.ant.tasks.ConverterTask" />
<taskdef name="verifyexport"
  classname="com.sun.javacard.ant.tasks.VerifyExpTask" />
<taskdef name="verifycap"
  classname="com.sun.javacard.ant.tasks.VerifyCapTask" />
<taskdef name="verifyrevision"
  classname="com.sun.javacard.ant.tasks.VerifyRevTask" />
<taskdef name="scriptgen"
  classname="com.sun.javacard.ant.tasks.ScriptgenTask" />
<typedef name="appletnameaid"
  classname="com.sun.javacard.ant.types.AppletNameAID" />
<typedef name="jcainputfile"
  classname="com.sun.javacard.ant.types.JCAInputFile" />
<typedef name="exportfiles"
  classname="org.apache.tools.ant.types.FileSet" />
```

## Library Dependencies

Table B-1 shows the libraries that are needed in your classpath if you are using the indicated feature. Alternatively, you can specify the classpath nested element for each task to put the required JAR files in the classpath during build execution.

**Table B-1 Library Dependencies**

| Libraries | Features |
|---|---|
| `converter.jar` and `offcardverifier.jar` | Creating CAP, EXP or JCA files.<br>Using `maskgen` to create a mask.<br>Dumping contents of an EXP file in a text file. |
| `offcardverifier.jar` | Verifying EXP files, CAP files and verifying binary compatibility between two versions of an export file. |
| `apdutool.jar` and `apduio.jar` | Sending an APDU script to `cref`. |
| `capdump.jar` | Dumping contents of a CAP file. |
| `scriptgen.jar` | Generating an APDU script from a CAP file. |

**Table B-1 (Cont.) Library Dependencies**

| Libraries | Features |
|---|---|
| apdutool.jar, apduio.jar and scriptgen.jar | Installing a CAP file in cref and generate resulting EEPROM image. |

# Ant Task Descriptions

The eleven Ant tasks provided in the Ant tasks bundle are provided to simplify the use of the development kit for Ant users. This section describes each of these Ant tasks and how to use them. Note that the JAR files for the tasks are expected to be in the system classpath, unless otherwise noted.

- APDUTool
- CapDump
- Capgen
- Converter
- DeployCap
- Exp2Text
- Maskgen
- 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-2 Parameters for APDUTool**

| Attribute | Description | Required |
|---|---|---|
| ScriptFile | Fully qualified path and name of the APDU script file. | Yes |
| CrefExe | Fully qualified path and name of 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 |

[1] The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java(TM) platform.

**Table B-2 (Cont.) Parameters for APDUTool**

| 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 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 any of the required elements are not supplied, if `apdutool.jar` and `apduio.jar` are not in the classpath, or if APDUTool returns an error code.

## Examples

To use these examples:

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-3 Parameters for CapDump**

| Attribute | Description | Required |
|---|---|---|
| CapFile | Fully qualified name of CAP file. | Yes |
| classpath | Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed. | No |
| dir | The directory in which to invoke the JVM software. | No |

## Errors

Execution of this task fails if CapFile element is not supplied, if `capdump.jar` is not in the classpath, or if CapDump returns an error code.

## Examples

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-4 Parameters for Capgen**

| Attribute | Description | Required |
|---|---|---|
| JCAFile | Fully qualified path and name of the input JCA file. | Yes |
| OutFile | Fully qualified path and name of the output CAP file. | No |
| classpath | Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed. | No |
| dir | The directory in which to invoke the JVM software. | No |
| nobanner | Set this element to `true` if you do not want the Capgen banner showing. | No |
| version | Prints Capgen version number. | No |

## Errors

Execution of this task fails if any of the required elements are not supplied, if `converter.jar` is not in the classpath, or if Capgen returns an error code.

## Examples

To use these examples:

1. Enter the following example code to run Capgen to generate the `mathDemo.cap` file from the `mathDemo.jca` file.

```
<target name="CapgenTarget" >
  <capgen
      JCAFile="${sample.output}/mathDemo.jca"
      outfile="${sample.output}/mathDemo.cap">
      </capgen>
</target>
```

2. Enter the following example code to run Capgen to generate a `mathDemo.cap` file from the `mathDemo.jca` file.

> **Note:**
>
> Classpath used in this example is referenced by the `classpath` refid.

```
<target name="CapgenTarget" >
  <capgen
      JCAFile="${sample.output}/mathDemo.jca"
      outfile="${sample.output}/mathDemo.cap">
      <classpath refid="classpath"/>
  </capgen>
</target>
```

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}/mathDemo.jca"/>
      <classpath refid="classpath"/>
  </capgen>
</target>
```

## Converter

Runs Converter to generate CAP, EXP and JCA files from a Java technology-based package. By default the Java Card platform converter creates CAP and EXP files for the input package. However, if any one of the CAP, JCA or EXP flags are enabled, only the output files enabled are generated.

**Table B-5 Parameters for Converter**

| Attribute | Description | Required |
|---|---|---|
| PackageName | Fully qualified name of the package being converted. | Yes |

**Table B-5 (Cont.) Parameters for Converter**

| Attribute | Description | Required |
|---|---|---|
| PackageAID | AID of the package being converted. | Yes |
| MajorMinorVersion | Major and Minor version numbers of the package, for example, 1.2 (where 1 is major version number and 2 is minor version number). | Yes |
| CAP | If enabled, tells the converter to create a CAP file. | No |
| EXP | If enabled, tells the converter to create a EXP file. | No |
| JCA | If enabled, tells the converter to create a JCA file. | No |
| ClassDir | The root directory of the class hierarchy. Specifies the directory where the converter 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 |
| 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 |

## 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 any of the required elements are not supplied, if `converter.jar` or `offcardverifier.jar` are not in the classpath, or if Converter returns an error code.

## Examples

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-6 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 any of the required elements are not supplied, if `apdutool.jar`, `apduio.jar` and `scriptgen.jar` are not in the classpath, or if APDUTool, Scriptgen or `cref` fail to execute.

## Examples

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-7    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 any of the required elements are not supplied, if `converter.jar` is not in the classpath, or if `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.sun.javacard.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.sun.javacard.samples.HelloWorld">
      <classpath refid="classpath"/>
  </Exp2Text>
</target>
```

## Maskgen

Runs Maskgen to generate a mask for `cref`, depending on the generator used (see details below).

**Table B-8 Parameters for Maskgen**

| Attribute | Description | Required |
|---|---|---|
| Generator | Tells Maskgen for which platform is the mask to be generated. Possible choices are a51, `cref`, and size. For details see Maskgen documentation in the Producing a Mask File from Java Card Assembly Files. | Yes |
| ConfigFile | Fully qualified path and name of generator specific configuration file. | No |
| DebugInfo | If enabled, tells Maskgen to generate location debug information for mask. | No |
| MemRefSize | Integer value that tells Maskgen what memory reference size to use in the mask. Two possible values for element are 16 and 32. Default value used by Maskgen is 32. | No |

**Table B-8 (Cont.) Parameters for Maskgen**

| Attribute | Description | Required |
|---|---|---|
| OutFile | Fully qualified path and name of the output mask file. If this element is not specified, the default file name is `a.out` and is generated in the directory where the Java VM is invoked. | No |
| classpath | Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed. | No |
| dir | The directory to invoke the Java VM in. | No |
| nobanner | Set this element to `true` if you do not want the Maskgen banner showing. | No |
| version | Prints Maskgen version number. | No |

## Parameters Specified As Nested Elements

The JCAInputFile parameters are specified as nested elements and use nested element JCAInputFile to specify names of input JCA files for Maskgen. Input JCA files are required to create a Mask file. The reason a standard FileSet to specify JCA file names is not used here is that Maskgen supports input file names that starts with an "@" symbol to specify an input file that contains a list of names of input JCA files. A file name that starts with "@" is not supported by any of the standard Ant types. See the description for JCAInputFile for details.

## Errors

Execution of this task fails if any of the required elements are not supplied, if `converter.jar` is not in the classpath or if `Maskgen` returns an error code.

## Examples

To use these examples:

1. Enter the following example code to run Maskgen to generate `mask.c` file from input JCA files specified in files `mask1.in` and `mask2.in`:

```
<target name="MasgenTarget" >
  <maskgen
      generator="cref"
      configfile="${maskDir}/mask.cfg"
      outfile="${crefDir}/common/mask.c" >
      <jcainputfile inputfile="@${maskDir}/mask1.in" / >
      <jcainputfile inputfile="@${maskDir}/mask2.in" / >
  </maskgen >
</target >
```

2. Enter the following example code to run Maskgen to generate `mask.c` file from input JCA files specified in files `api.in` and installer and `helloworld` JCA files:

```
<target name="MasgenTarget" >
  <maskgen
      generator="cref"
```

```
        configfile="${maskDir}/mask.cfg"
        outfile="${crefDir}/common/mask.c" >
        <jcainputfile inputfile="@${maskDir}/api.in" / >
        <jcainputfile inputfile="${jcaDir}/installer.jca" / >
        <jcainputfile inputfile="${jcaDir}/helloworld.jca" / >
    </maskgen >
  </target >
```

3. Enter the following example code to run Maskgen without specifying an output file and classpath.

> **Note:**
>
> Maskgen generates the file `a.out` in the directory in which Java VM was invoked.

```
<target name="MasgenTarget" >
  <maskgen
      generator="cref"
      configfile="${maskDir}/mask.cfg"
      outfile="${crefDir}/common/mask.c" >
      <jcainputfile inputfile="@${maskDir}/api.in" / >
      <jcainputfile inputfile="${jcaDir}/installer.jca" / >
      <jcainputfile inputfile="${jcaDir}/helloworld.jca" / >
      <classpath refid="classpath"/>
  </maskgen >
</target >
```

## Scriptgen

Runs Scriptgen to generate an APDU script file from a CAP file.

**Table B-9 Parameters for Scriptgen**

| Attribute | Description | Required |
|---|---|---|
| CapFile | Fully qualified path and name of the input CAP file. | Yes |
| HashFile | Fully qualified path and name of the verifier-generated file that contains the hashes for all of 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 |
| 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 |

**Table B-9 (Cont.) Parameters for Scriptgen**

| Attribute | Description | Required |
|---|---|---|
| version | Prints Scriptgen version number. | No |

## Errors

Execution of this task fails if any of the required elements are not supplied, if `scriptgen.jar` is not in the classpath or if Scriptgen returns an error code.

## Examples

To use these examples:

1. 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-10 Parameters for VerifyCap**

| Attribute | Description | Required |
|---|---|---|
| CapFile | Fully qualified path and name of CAP file that is to be verified. | Yes |
| PkgName | Fully qualified Name of the package inside the CAP file for which the CAP file was generated. | No |
| noWarn | If enabled, tells the verifier not to output any warning messages. | No |
| Verbose | If enabled, enables verbose verifier output. | No |

**Table B-10 (Cont.) Parameters for VerifyCap**

| 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 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 any of the required elements are not supplied, if `offcardverifier.jar` is not in the classpath, or if Verifier returns an error code.

## Examples

To use these examples:

1. 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-11 Parameters for VerifyExp**

| Attribute | Description | Required |
|---|---|---|
| noWarn | If enabled, tells the verifier not to output any warning messages. | No |

**Table B-11    (Cont.) Parameters for VerifyExp**

| Attribute | Description | Require d |
|---|---|---|
| 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 no EXP file is specified or if more than one EXP file is specified, if `offcardverifier.jar` is not in the classpath, or if Verifier returns an error code.

## Examples

To use these examples:

1. Enter the following example code to run the Java Card platform off-card verifier to verify `HelloWorld.exp` file:

```
<target name="VerifyExpTarget" >
  <verifyExp
      <exportfiles file="${samples.helloworld.output}/HelloWorld.exp" />
      <classpath refid="classpath"/>
  </verifyExp>
</target>
```

# VerifyRev

Runs off-card Java Card platform verifier to verify binary compatibility between two versions of an EXP file. Java Card platform off-card verifier is invoked in a separate instance of Java VM.

**Table B-12    Parameters for VerifyRev**

| Attribute | Description | Require d |
|---|---|---|
| noWarn | If enabled, tells the verifier not to output any warning messages. | No |

**Table B-12 (Cont.) Parameters for VerifyRev**

| Attribute | Description | Required |
|---|---|---|
| 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 or if less or more than two EXP files are specified, if `offcardverifier.jar` is not in the classpath, or if Verifier returns an error code.

## Examples

To use these examples:

1. 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" >
  <verifyExp
      <exportfiles file="${samples.helloworld.output}/HelloWorld.exp" />
      <exportfiles file="${samples.helloworld.output.new}/HelloWorld.exp" />
      <classpath refid="classpath"/>
  </verifyExp>
</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-13 Parameters for AppletNameAID**

| Attribute | Description | Required |
|---|---|---|
| appletname | Fully qualified name of the Java Card applet. | Yes |
| aid | AID (Application Identifier) of the Java Card applet. | Yes |

## Example

To use these examples:

1. 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-14 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\mathDemo.in" />
```

## ExportFiles

This type is actually the Ant FileSet type. It is used to specify a group of export files for the off-card verifier. For details, see Apache Ant documentation for FileSet type.

### Examples

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 applet applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies.

A unique AID is associated with each applet class in an applet application module. In addition, a unique AID is assigned to each applet instance during installation. This applet instance AID is used by an off-card client to select the applet instance for APDU communication sessions.

Applet instance URIs are constructed from their applet instance AID using the "aid" registry-based namespace authority as follows:

```
//aid/<RID>/<PIX>
```

where `<RID>` (resource identifier) and `<PIX>` (proprietary identifier extension) are components of the AID.

**APDU**

an acronym for Application Protocol Data Unit as defined in ISO 7816-4.

**APDU-based application environment**

consists of all the functionalities and system services available to applet applications, such as the services provided by the applet container.

**applet**

within the context of this document, a Java Card applet, which is the basic component of applet-based applications and which runs in the APDU application environment.

**applet application**

an application that consists of one or more applets.

**applet container**

contains applet-based applications and manages their lifecycles through the applet framework API. Also provides the communication services over which APDU commands and responses are sent.

**applet framework**

an API that enables applet applications to be built.

**applicable security requirements**

the security requirements instance, application-assigned or card manager-assigned, that applies for a particular mode of communication. See security requirements.

**application assembler**

takes the output of the application developer and ensures that it is a deployable unit. Thus, the input of the application assembler is the application classes and resources, and other supporting libraries and files for the application. The output of the application assembler is an application archive.

**application-defined event**

an event that an application may define in its own namespace and may be the only one allowed to fire.

**application-defined service**

a service that an application may define in its own namespace and may be the only one allowed to register.

**application descriptor**

see descriptor.

**application framework class loader**

a direct child of the extension library class loader, in charge of loading application framework libraries shared among a restricted set of application groups.

**application group**

a set of one or more applications executing in a common group context.

**application-managed authentication**

authentication that is programmatically triggered by an application's code based on some business logic.

**application module class loader**

a direct child in the class loader delegation hierarchy of either a group library class loader or of the classic library class loader, depending on the type of application model, in charge of loading the application module classes.

**application protection domain**

the set of permissions effectively granted to an application, that results from the combination of permissions granted by the platform security policy and the permissions granted by the card management security policy.

**application security policy**

a role-based security policy defined for a specific application and for which all the logical user and client security roles have been mapped to actual user identities and client application identities or characteristics on the platform to which the application is deployed.

**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.

**authentication**

the process of establishing or confirming an application or a user as authentic using some sort of credentials

**authenticator**

an authentication service that can be invoked both by applications for application-managed authentication and by the web container for container-managed authentication.

**authorization**

the process of allowing access to those resources by entities (applications or users) that have been granted authority to use them.

**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.

**bootstrap class loader**

the root of the class loader delegation hierarchy in charge of loading the Java Card RE system classes.

**bytecode**

machine-independent code generated by the compiler and executed by the Java virtual machine.

**card holder**

the primary user of a smart card.

**card holder-facing client**

a client that may directly and safely interact with the card holder. A card holder-facing client may typically be local, co-hosted on the card-hosting device, or in close proximity to the card.

**card holder user**

a user whose identity may be assumed by the card holder.

**card manager**

the on-card application to download and install applications and libraries. The card manager receives executable binary and metadata from the off-card installer, 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.

**card management facility**

the Java Card platform layer responsible for securely adding and removing application code and instances onto the platform.

**card management security policy**

a permission-based security policy that is defined by a card management authority and that grants some permissions to an application or group of applications in accordance with the operational environment in which the application or group of applications is deployed.

**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.

**classic applet**

applets with the same capabilities as those in previous versions of the Java Card platform and in the Classic Edition.

**classic applet container mutex object**

the object that is used by the Java Card RE to synchronize all concurrent accesses to a classic applet's code in order to guarantee its thread safety.

**Classic Edition**

the Classic Edition is based on an evolution of the Java Card Platform, Version 2.2.2 and is backward compatible with it, targeting resource-constrained devices that solely support applet-based applications.

**classic library**

a Java programming language package that does not contain any non-abstract classes that extend the class `javacard.framework.Applet`. A classic applet application comprises a Java programming language package that contains one or more non-abstract classes that extend the `javacard.framework.Applet` class.

**classic library class loader**

a direct child of the shareable interface class loader in charge of loading classic library classes.

**classic SIO proxy**

see classic SIO synchronization proxy.

**classic SIO synchronization proxy**

an object that implements a shareable interface of a classic applet application and that synchronizes with all other concurrent accesses to the classic applet application before delegating to the actual SIO. An SIO synchronization proxy is returned to each client of the classic applet application that requests access to that shareable interface.

**class loader**

a Java Card RE component that defines and enforces a different class namespace for the classes it loads.

**class loader delegation hierarchy**

the hierarchy of class loaders that enforces code isolation among applications while allowing for sharing of system and library code.

**client application**

an on-card application that uses services provided by other applications (server applications).

**converter**

a piece of software that preprocesses all of the Java programming language class files of a classic applet application that make up a package, and converts the package into a standalone classic applet application module distribution format (CAP file). The Converter also produces an export file.

**currently active context**

when an object instance method is invoked, an owning context of the object becomes the currently active context for that particular thread of execution.

**currently active namespace**

corresponds to the application owner identifier of the active context set upon entry into the group context for a particular thread of execution.

**currently selected applet**

the applet container keeps track of the currently selected Java Card applet. Upon receiving a SELECT FILE command with this applet's AID, the applet container makes this applet the currently selected applet. The applet container sends all APDU commands to the currently selected applet.

**declarative security**

a means of expressing an application's security structure, including roles, access control, and authentication requirements in a form external to the application, such as in the deployment descriptor of a web application.

**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.

**deployer**

The deployer takes one or more application archive files provided by an application developer and deploys the application into a card in a specific operational environment. The operational environment includes other installed applications and libraries, as well as standard bodies-defined frameworks. The deployer must resolve all the external dependencies declared by the developer.

The deployer is an expert in a specific operational environment. For example, the deployer is responsible for mapping the security roles defined by the application developer to the users that exist in the operational environment where the application is deployed.

**deployment unit**

entity that can be distributed, deployed and installed on the Java Card platform.

**deployment descriptor**

see descriptor.

**descriptor**

a document that describes the configuration and deployment information of an application. A deployment descriptor conveys the elements and configuration information of an application between application developers, application assemblers, and deployers. A runtime descriptor describes the configuration and deployment information of an application that are specific to an operating environment to which the application is to be deployed.

**distribution format**

structure and encoding of a distribution or deployment unit intended for public distribution.

**distribution unit**

see deployment unit.

**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.

### event

an object that encapsulates some occurring condition or situation. In the context of the event notification facility, an event is a shareable interface object that an application (event-producing application) uses to notify its clients (event-consuming applications) of an occurring condition.

### event consuming application

an application that registers for notification of events fired by an event producing application.

### event listener

an object that is registered to handle events when they occur. In the context of the event notification facility, an event listener is an object that a client application (event-consuming application) registers and uses to handle SIO-based events an application (event-producing application) produces.

### event notification facility

a Java Card RE facility (or subsystem) that is used for event-driven inter-application communications.

### event notification listener

see event listener.

### event producing application

an application that fires events.

### event registry

the core component of the event notification facility. The event registry is used for registering for notification of events and for notifying of events.

### event URI

a URI that uniquely identifies an event produced by an event-producing application.

### 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.

**extended applet**

an applet with extended and advanced capabilities (compared to a classic applet) such as the capabilities to manipulate `String` objects and open network connections.

**extension library**

library that extends the functionality of the platform.

**extension library class loader**

a direct child of the shareable interface class loader in the class loader delegation hierarchy in charge of loading extension 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 of a classic applet application are represented in an export file. For a classic library package, all classes, interfaces, their constructors, methods, and fields of an application that can be accessed from another application according to the Java programming language access control semantics, as defined by the *Java Language Specification* are listed in the export file.

**file permissions mode**

an attribute of a file system object that indicates whether read or write operation on the object are permitted or denied.

**filter**

a web application component that is used to transform the content or header information of HTTP requests or responses.

**finalization**

the process by which a Java virtual machine (VM) 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.

**garbage collection**

the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.

**global array**

an applet environment array objects accessible from any context.

**global authentication**

the scope of a user authentication that can be tracked globally (card-wide). Global authentication is restricted to card-holder-users. Authorization to access resources protected by a globally authenticated card-holder-user identity is granted to all users.

**group context**

protected object space associated with each application group and Java Card RE. All objects owned by an application belong to the context of the application group.

**group-library class loader**

a direct child of the extension library class loader in charge of loading the libraries private to an application group. Libraries, private to different application groups, are loaded by distinct group library class loaders, one per web or extended applet application group.

**heap**

a common pool of free memory in volatile and persistent spaces usable by a program. A part of the computer's memory used for dynamic memory allocation, in which blocks of memory are used in an arbitrary order.

The Java Card virtual machine's volatile heap is typically garbage collected on demand and on card tear.

The Java Card virtual machine's persistent heap is typically garbage collected on a less frequent basis. Memory associated with objects allocated from the persistent heap are not necessarily reclaimed.

**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

items that are not externally visible to other applications on the card. See also *externally visible*.

### inter-application communication facility

see event notification facility, service facility.

### 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 APDU application environment. It provides a mechanism for a client application to invoke a method on a remote object of an applet application on the card.

### Java Card Runtime Environment (Java Card RE)

consists of the Java Card virtual machine and the associated native methods.

### Java Card Virtual Machine (Java Card VM)

a subset of the Java virtual machine, which is designed to be run on smart cards and other resource-constrained devices. The Java Card VM acts an engine that loads Java class files and executes them with a particular set of semantics.

### 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 application group 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.

**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.

**locally accessible web application**

an application that may interact with the card holder.

**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.

**mode (communication)**

designates the type or protocol of communication (HTTPS, SSL/TLS, SIO...) and the mode of operation (client or server) that characterizes a communication endpoint.

**module (application)**

the logical unit of assembly of web or applet-based application. The components of a web application are assembled into a web application module. The components of an applet application are assembled into a applet application module.

**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.

**named permission**

a permission that has a name but no actions list; the named permission is either granted or not. A named permission typically protects a function or functionality.

**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 Card Manger does not load applications containing native methods.

**non-card holder-facing client**

a client that does not directly interact with the card holder, but interacts with some other-users such as remote administrators. A non-card holder-facing client may typically be a remote system that may interact with the card through the network to which the card-hosting device itself is connected.

**non-volatile memory**

memory that is expected to retain its contents between card tear and power up events or across a reset event on the smart card device.

**normalization (classic applet)**

the process of transforming and repackaging a Java application packaged for the Java Card Platform, Version 2.2.2, for deployment on both the Java Card Platform, Version 3, Connected Edition and the Java Card Platform, Version 3, Classic Edition.

**normalization (URI)**

the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI.

**Normalizer**

a software tool that allows Java applications packaged for Version 2.2.2 to be transformed and then repackaged for deployment on the Classic Edition.

**object owner**

the applet instance context or web application context or the Java Card RE context which was the currently active context when the object was instantiated.

**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.

**off-card client**

see off-card client application.

**off-card client application**

an application that is not resident on the card, but runs at the request of a user's actions.

**off-card installer**

the off-card application that transmits the application and library executables to the card manager application running on the card.

**off-card proxy generator**

a program or tool used to generate classic SIO synchronization proxies prior to packaging and deploying a classic applet application.

**on-card client**

see client application.

**origin logical channel**

the logical channel in the APDU application environment on which an APDU command is issued.

**other user**

a user other than a card holder user, such as a remote card administrator.

**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.

**permission**

an object that represents access to specific protected resources, such as security-sensitive system resources, or application resources, such as services provided by applications. Permissions are instances of subclasses of the `Permission` class. A permission has a name and may have an actions list.

**permission actions list**

an attribute of a permission used to designate those actions for which the resources designated by the target name are protected.

**permission-based security**

measures defined by a permission-based security policy that restrict access to protected system and library resources.

### permission-based security policy

a security policy that maps some of the characteristics of an application requesting access to a protected resource to a set of permissions granted to the application.

### permission name

an attribute of a permission used to designate a protected function or resource, or a set thereof.

### permission target name

the name attribute of a permission object (permission name) that designates the resource or set of resources that are protected with that permission.

### permission type

a type defined by a permission class.

### persistent object

persistent objects and their values persist from one card session to the next, indefinitely. 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).

### platform event

a well-defined event fired by the platform. Examples are clock resynchronization events.

### platform protection domain

a set of permissions granted to an application or group of applications by the platform security policy. A platform protection domain is defined by two sets of permissions: a set of included permissions that are granted and a set of excluded permissions that are denied and can never be granted.

### platform security policy

the permission-based security policy that maps application models to sets of permissions granted to applications implementing these application models. For each of the application models, the platform security policy guarantees the consistency and integrity of the applications implementing the application model.

### principal

an entity that can be authenticated by an authentication protocol. A principal is identified by a *principal name* and authenticated by using *authentication data*. The content and format of the principal name and the authentication data depend on the authentication protocol.

### programmatic security

a means for a security aware application to express the security model of the application when declarative security alone is not sufficient.

### protected content

see protected resource.

### protected resource

an application or system resource that is protected by an access control mechanism.

### protection domain

a set of permissions granted to an application or group of applications.

### RAM (random access memory)

temporary working space for storing and modifying data. RAM is non-persistent memory; that is, the information content is not preserved when power is removed from the memory cell. RAM can be accessed an unlimited number of times and none of the restrictions of EEPROM apply.

### reachability disrupting object

a special object that prevents the promotion of a volatile object to become a persistent object. If a volatile object is referenced by a persistent object, which is not a reachability disrupting object, or by a root of persistence, the volatile object is automatically promoted and becomes a persistent object. An example of reachability disrupting object is a `TransientReference` object.

### reference implementation

a fully functional and 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 *Runtime Environment Specification, Java Card Platform, v3.0.5, Classic Edition.*

### 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.

**remote user**

an user whose identity may be assumed by a remote entity, such as a remote card administrator.

**restartable task**

an object implementing the `Runnable` interface that has been registered for recurrent execution over card sessions. A task executes in its own thread.

**restartable task registry**

a Java Card RE facility that is used for registering tasks for recurrent execution over card sessions.

**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).

**role (security)**

an abstract notion used by an application developer in an application that can be mapped by the deployer to a user, or group of users, in a security policy domain.

**role-based security**

measures defined by a role-based security policy that restrict access by clients or by users to protected application resources.

**role-based security policy**

a security policy that maps some of the characteristics of an application requesting access to protected resources, such as its identity and the identity of the user on behalf of whom the access is requested to roles permitted to access the protected resources.

**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.

**root URI**

a URI that identifies the root of an application's namespace for a particular scheme. Examples are an application's service root URI, an application's event root URI.

**runtime descriptor**

see descriptors.

**runtime environment**

see Java Card Runtime Environment (Java Card RE).

**secure port redirector**

a web application container that redirects HTTP requests for protected content sent over unsecure connections to the secure port over which that content can be served. Protected content must be served only over a secure port.

**security constraint**

a declarative way of defining the protection of web content. A security constraint associates authorization and or user data constraints with HTTP operations on web resources.

**security policy domain**

the scope over which security policies are defined and enforced by a security administrator of the security service. A security policy domain is also sometimes referred to as a *realm*.

**security policy**

designates the protected resources that can be accessed by individual applications or groups of applications. These protected resources may be security-sensitive system resources or application resources such as services provided by other applications.

**security requirements**

the required security characteristics for a particular secure communication being established by either an application or by the web container on behalf of a web application.

**service**

a shareable interface object that a server application uses to provide a set of well-defined functionalities to its clients.

**service facility**

a Java Card RE facility (or subsystem) that is used for inter-application communications.

### service factory

an object that the Java Card RE invokes to create a service - on behalf of the server application that registered that service - for a client application that looked up the service.

### service registry

the core component of the service facility. The service facility is used for registering and looking up services.

### shareable interface

an interface that defines a set of shared methods. These interface methods can be invoked from an application in one group context when the object implementing them is owned by an application in another group context.

### shareable interface class loader

the direct child of the bootstrap class loader in the class loader delegation hierarchy in charge of loading publicly exposed shareable interfaces.

### shareable interface object (SIO)

an object that implements the shareable interface.

### shareable interface object-based service

see service.

### smart card

a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smart cards carry both processing power and information. They do not require access to remote databases at the time of a transaction.

### 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.

### standard event

a standard event with a well-defined semantic that an application may fire. Examples are standard application lifecyle events such as application creation and deletion events, and standard resource lifecyle events such as resource creation and deletion events.

### standard service

a standard service with a well-defined interface that an application may provide and register. Examples are authenticators - authentication 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.

### thread's active context

when an object instance method is invoked, the owning context of the object becomes the currently active context for that particular thread of execution. Synonymous with currently active context.

### transaction

an atomic operation in which the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

### transaction facility

a Java Card RE facility that enables an application to complete a single logical operation on application data atomically, consistently and durably within a transaction.

### transient object

the state of transient objects do not persist from one card session to the next, and are reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions.

### transferable classes

classes whose instances can have their ownership transferred to a context different from their currently owning context. Transferable classes are of two types:

**Implicitly transferable classes** - Classes whose instances are not bound to any context (group contexts or Java Card RE context) and can, therefore, be passed and shared between contexts without any firewall restrictions. Examples are `Boolean` and literal `String` objects.

**Explicitly transferable classes** - Classes whose instances must have their ownership explicitly transferred to another application's group context in order to be accessible to that other application. Examples are arrays and newly created `String` objects.

### transfer of ownership

a Java Card RE facility that allows for an application to transfer the ownership of objects it owns to an other application. Only instances of transferable classes can have their ownership transferred.

### trusted client

an on-card or off-card application client that an on-card application trusts on the basis of credentials presented by the client.

### trusted client credentials

credentials that an on-card application uses to ascertain the identity of clients it trusts.

### 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 ensures that the binary representation of the application or library is structurally correct.

### volatile memory

memory that is not expected to retain its contents between card tear and power up events or across a reset event on the smart card device.

### volatile object

an object that is ideally suited to be stored in volatile memory. This type of object is intended for a short-lived object or an object which requires frequent updates. A volatile object is garbage collected on card tear (or reset).