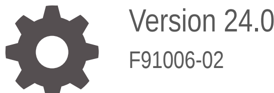
Java Card

Developer Kit Simulator User Guide



Version 24.0

F91006-02

Java Card Developer Kit Simulator User Guide, Version 24.0

F91006-02

Copyright © 2024, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

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, software documentation, data (as defined in the Federal Acquisition Regulation), 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:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

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.

Oracle®, Java, MySQL and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

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

Audience xi
Documentation Accessibility xi
Related Documents xii
Documentation and Support xii
Third-Party Web Sites xii
Conventions xii

Part I Setup, Samples, and Tools

1 Introduction

Java Card Platform Architecture 1-1
  Java Card 3.2 1-2
Communication Options Between Host and Java Card Simulator 1-3
GlobalPlatform 1-3

2 Installation

Install and Setup the Java Card Development Kit Simulator 2-1
  Before Installing the Java Card Development Kit Simulator 2-1
    Build and Install the OpenSSL Library 2-2
    Install and Setup the PCSC-Lite and IFD Handler 2-6
  Installing the Java Card Development Kit Simulator 2-7
  Installed Files and Directories 2-8
  Verifying the System Variables 2-8
Configuring the Java Card Development Kit Simulator 2-8
Uninstalling the Java Card Development Kit Simulator 2-9
Set up the Eclipse Java Card Plug-in 2-9
  Installing the Eclipse Plug-in 2-10
  Configuring Sample_Platform and Sample_Device 2-11
    Configuring Sample_Platform 2-11

Configuring Sample_Device 2-11
Configuring the Java Card Tools Path 2-12

3 Running the Simulator and the Samples

Java Card Development Kit Simulator Command Line 3-1
How to Run the Samples 3-2
Running the Samples in Eclipse 3-2
Running the Samples from the Command Line 3-5
Running the Included Samples 3-6
HelloWorld Sample 3-6
Running the HelloWorld Sample in Eclipse 3-6
Running the HelloWorld Sample from the Command Line 3-7
Channels Sample 3-7
Running the Channels Sample in Eclipse 3-8
Running the Channels Sample from the Command Line 3-8
Utility Sample 3-9
Running the Utility Sample in Eclipse 3-10
Running the Utility Sample from the Command Line 3-10
Wallet Sample 3-11
Running the Wallet Sample in Eclipse 3-11
Running the Wallet Sample from the Command Line 3-12
ObjectDeletion Sample 3-12
Running the ObjectDeletion Sample in Eclipse 3-12
Running the ObjectDeletion Sample from the Command Line 3-13
SignatureMessageRecovery Sample 3-13
Running the SignatureMessageRecovery Sample in Eclipse 3-14
Running the SignatureMessageRecovery Sample from the Command Line 3-15
Message Recovery Order of Operations for Signing 3-15
Message Recovery Order of Operations for Verifying 3-15
ArrayViews Sample 3-16
Running the ArrayViews Sample in Eclipse 3-16
Running the ArrayViews Sample from the Command Line 3-17

4 Debugging Applications

Debugger Architecture 4-1
Running the Debug Proxy from the Command Line 4-1
Debug Proxy Options 4-2
Debugging the HelloWorld Sample from the Command Line 4-3

5 Packaging and Deploying Your Application

Overview of Packaging and Deploying Applications 5-1
Installer Components and Data Flow 5-1
Running Client Application for Applet Management 5-2
Downloading CAP Files and Creating Applets 5-3
Using the On-card Installer for Deletion 5-3
    How to Send a Deletion Request 5-4

Part II Programming With the Development Kit

6 Using Object, CAP File and Applet Deletion

Object Deletion Mechanism 6-1
    Requesting the Object Deletion Mechanism 6-1
    Object Deletion Mechanism Usage Guidelines 6-2
CAP File and Applet Deletion 6-2
    Developing Removable CAP File 6-3
    Writing Removable Applets 6-3
        The AppletEvent.uninstall Method 6-3

7 Working with Logical Channels

Dual Interface Cards 7-1
Applets and Logical Channels 7-1
    Non-MultiSelectable Applets 7-1
The MultiSelectable Interface 7-2
    Selection for MultiSelectable Applets 7-2
    Deselection for MultiSelectable Applets 7-3
Writing Applets for Concurrent Logical Channels 7-3
    MultiSelectable Applet Example 7-4
    Handling Channel Information on APDU Commands 7-6
        Interindustry Space 7-7
        Proprietary Java Card Technology Space 7-7
        Logical Channels 7-8
        APDU Command Type Identification 7-8
    Writing ISO/IEC 7816-4:2013 Compliant Applets 7-9
        ISO/IEC 7816-4:2013 Compliant Applet Example 7-9
    Non-MultiSelectable Applets and Shareable Objects 7-10
    ISO/IEC 7816-4:2013 Specific APDU Commands for Logical Channel Management 7-11
        MANAGE CHANNEL OPEN 7-11

MANAGE CHANNEL CLOSE 7-12
SELECT FILE 7-13

8 Using Extended APDU

Extended APDU Nominal Cases 8-1
Extended APDU Format 8-1
Extended APDU Limits 8-2
Creating an Applet That Can Send and Receive Extended Length APDUs 8-3

9 Java Card Development Kit Accessibility Information

Access to Java Card Development Kit Support 9-1
Java Card Development Kit Features that Support Accessibility 9-1
Keyboard Navigation 9-1
Documentation Accessibility Features 9-2

Part III Java Card Eclipse Plug-in

10 Using the Java Card Eclipse Plug-in

Creating a Java Card Project Using the New Java Card Project Wizard 10-1
Changing the Runtime Environment for the Java Card Project 10-1
Creating a Java Card Applet Using the Default Source Template 10-2
Creating a CAP File in a Java Card Project 10-2
Managing CAP File Configurations 10-3
Adding a Java Card Package to a CAP File 10-3
Managing the Java Card Package 10-4
Adding a Java Card Applet to a Java Card Package 10-4
Managing Java Card Applets 10-5
Adding a Java Card Static Resource to a CAP File 10-5
Managing Java Card Static Resources 10-6
Managing Applets and Sending APDU Commands 10-6
Debugging a Java Card Applet in Eclipse Plug-in 10-8
Debug Configuration 10-9
Debugging HelloWorld Sample from Eclipse 10-9

Part IV Appendices

A Annex - Using Cryptography Extensions

Overview of Using Cryptography Extensions A-1
Supported Cryptography Classes A-2
Instantiating the Classes A-9

B Annex - Application Management

Supported GlobalPlatform Features B-1

Glossary

Index

Abstract

This document describes how to use the Java Card Development Kit for the Java Card Platform.

List of Figures

1-1	Java Card Architecture	1-2
3-1	Execution Environments	3-5
4-1	Debugging Architecture	4-1
5-1	Installed Components Data Flow	5-2

List of Tables

3-1 Command Line Parameters 3-1

8-1 Extended APDU Format 8-1

A-1 Algorithms Implemented by the Cryptography Classes A-3

B-1 Mapping terminology B-1

Preface

This document describes how to use the Java Card Development Kit to develop Java Card applications named applets.

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

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

Audience

This *Development Kit User Guide* is written for developers who are creating applets using the *Application Programming Interface, Java Card Platform, Version 3.2* and also for developers who are considering creating a vendor-specific framework based on the Java Card specifications.

Before You Read This Document

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

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

Information on Java Card technology, including access to the latest Java Card Development Kit downloads, is available at https://www.oracle.com/java/java-card/.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `https://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 `https://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `https://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

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

- *Java Card Platform Application Programming Interface Specification, Classic Edition, Version 3.2*
- *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.2*
- *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.2*
- *Java Card Platform Options List*
- *Off-Card Verifier for the Java Card Platform White Paper*
- *ISO 7816-4:2013 Specification*
- *GlobalPlatform Card Specification v2.3.1*
- *GlobalPlatform Secure Channel Protocol '03' - Amendment D v1.2*
- *GlobalPlatform Card API (org.globalplatform) v1.6*

Documentation and Support

These web sites provide additional resources:

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

Third-Party Web Sites

Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Conventions

The following text conventions are used in this document:

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

Part I

Setup, Samples, and Tools

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

- Introduction
- Installation
- Running the Simulator and the Samples
- Debugging Applications
- Packaging and Deploying Your Application

1 Introduction

The Java Card Development Kit is a suite of components and tools for designing implementations of Java Card technology and developing applets based on the Java Card API Specifications.

It is available as three independent downloads:

- The Java Card Development Kit Tools are used to convert and verify Java Card applications. The Tools can be used with products based on version 3.2 of the Java Card specifications, and can also be used with products based on versions 3.0.4, 3.0.5 and 3.1 of the Java Card Platform specifications, Classic Edition.
- The Java Card Development Kit Simulator offers a runtime reference to Java Card applications. It implements the version 3.2 of the Java Card specifications.
- The Java Card Development Kit Eclipse Plug-in offers an easy path for developing, testing and debugging Java Card applications.

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

This User Guide is dedicated to the Java Card Development Kit Simulator and Java Card Development Kit Eclipse Plug-in bundles. The Java Card Development Kit Tools bundle has its own dedicated User Guide.

For detailed information on bundles content, refer to the *Java Card Development Kit Tools Release Notes* and *Java Card Development Kit Simulator Release Notes*.

The Java Card Development Kit Simulator is only designed as an example of the functional behavior of a Java Card runtime.

Note:

It is not intended to operate in a production environment (and under the threats typically associated with such an environment).

This chapter contains the following sections:

- Java Card Platform Architecture
- Communication Options Between Host and Java Card Simulator
- GlobalPlatform

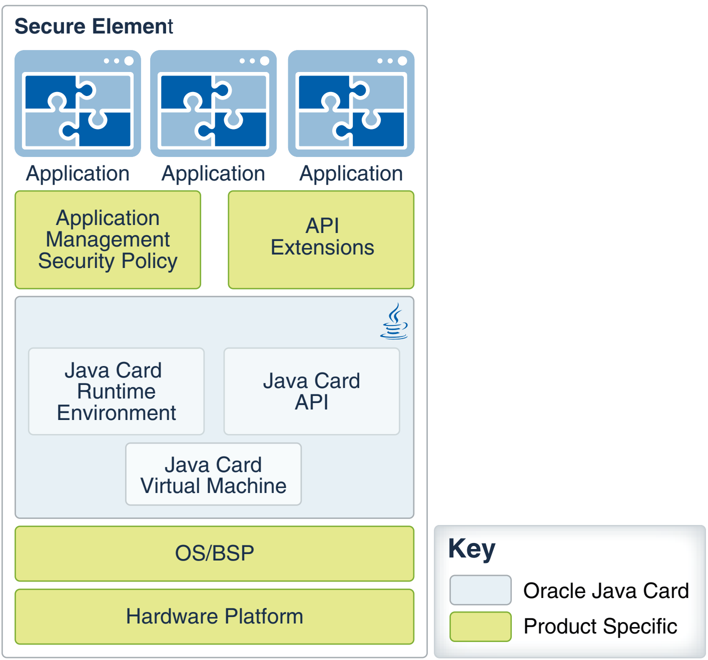
Java Card Platform Architecture

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

Applets that run on the Classic Edition are sometimes referred to as classic applets.

The Java Card Platform, Version 3.2 architecture illustrated below is built on the Classic Java Card VM. It preserves backward compatibility with Java Card applets written for earlier versions.

Figure 1-1 Java Card Architecture



This development kit includes an implementation of the Java Card RE, which stands for the simulator throughout this book. It is invoked on the command line with `jcsw.exe` on Windows, or `jcsl` on Linux.

Java Card 3.2

The Java Card Development Kit Simulator supports version 3.2 of the Java Card Platform specifications.

The document Java Card Platform Options List describes the list of options available to implement a Java Card Platform, based on the Java Card Specifications. The following Java Card optional features and packages are supported by the Java Card Development Kit Simulator:

- Integer support
- Extended length APDU
- Encodings for 4 or 20 Logical Channels
- Object Deletion
- Extended CAP files
- Sensitive arrays
- Key Derivation Function (package `javacardx.security.derivation`)

- Sensitive Result (package `javacardx.security`)
- ByteBuffer (package `javacardx.framework.nio`)
- StringUtil (package `javacardx.framework.string`)
- Montonic Counter (package `javacardx.security.util`)
- Certificate parsing (package `javacard.security.cert`)
- BCDUtil, BigNumber and ParityBit classes (package `javacardx.framework.math`)
- SysTime, TimeDuration classes (package `javacardx.framework.time`)
- TLV (package `javacardx.framework.tlv`)

Communication Options Between Host and Java Card Simulator

This section desribes the communication options between the host and the Java Card Simulator

The Java Card Simulator receives messages over network sockets (see the Java Card Development Kit Simulator Command Line section for information on how to configure the port number).

- Communication from client software running on Windows
  - The client application could use the java.smartcardio module in combination with the`%JC_HOME_SIMULATOR%\client\COMService\socketprovider.jar` library to communicate directly over IP sockets
- Communication from client software running on Linux
  - The client application could use the java.smartcardio module in combination with the `$JC_HOME_SIMULATOR/client/COMService/socketprovider.jar` library to communicate directly over IP sockets
  - Alternatively, the client application could use PCSC-Lite in combination with our IFDHandler (`$JC_HOME_SIMULATOR/drivers/IFDHandler/libjcsdkifdh.so`) to send APDUs over PC/SC.
    These APDUs will be automatically forwarded by the IFDHandler to the simulator over IP socket

For more details on how to use `socketprovider.jar` and `java.smartcardio`, refer to examples provided in the section Running Samples).

Details can be found in the script / batch files of the examples (for command line usage) and in the Eclipse launch configuration files as well.

For documentation of Java module `java.smartcardio` refer to:

https://docs.oracle.com/en/java/javase/17/docs/api/java.smartcardio/module-summary.html

GlobalPlatform

The Java Card SDK supports `GlobalPlatform` specification version 2.3.1 for application management.

Please refer to the Annex B of this document for the complete set of `GlobalPlatform` supported features.

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. It also describes how to install and setup the Eclipse Plug-in.

The development kit is available as three independent downloads:

- The Java Card Development Kit Tools are used to convert and verify Java Card applications.
- The Java Card Development Kit Simulator provides a runtime reference for Java Card applications.
- The Java Card Development Kit Eclipse Plug-in offers an easy path for developing, testing, and debugging Java Card applications.

This chapter contains the following sections:

- Install and Setup the Java Card Development Kit Simulator
- Configuring the Java Card Development Kit Simulator
- Uninstalling the Java Card Development Kit Simulator

Install and Setup the Java Card Development Kit Simulator

The Java Card Development Kit Simulator offers a developing, testing, and debugging reference for Java Card applications. It includes a Java Card simulation environment that supports the latest Java Card 3.2 Specification and can also run applications written for earlier releases.

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

- Before Installing the Java Card Development Kit Simulator
- Installing the Java Card Development Kit Simulator
- Installed Files and Directories
- Verifying the System Variables

Before Installing the Java Card Development Kit Simulator

This product is targeted for use on a PC running on the following operating systems

- Microsoft Windows versions 10 or 11
- Ubuntu Linux 20.04 LTS

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

- **Java Card Development Kit Tools –** Install the latest version and refer to the Install and Setup the Java Card Development Kit Tools chapter of its User Guide for further instructions.
- **OpenSSL** - The OpenSSL 3.0 library (32-bit) is required for the Java Card Development Kit Simulator to work. The Simulator has been tested using OpenSSL 3.0.12. Additionally, the legacy library is needed for the support of single DES algorithm. See more details in the Build and Install the OpenSSL Library chapter.
- **Java Development Kit (JDK)** - The tools in this development kit were tested with Oracle JDK 17 (64 bit version) and OpenJDK 17 (64 bit version). You can download and install the JDK release according to the instructions on the website:

  `https://www.oracle.com/technetwork/java/javase/downloads`
- **Eclipse IDE (optional)** - Using the Eclipse IDE as your development environment is recommended, although you can also run the samples and the development kit tools from the command line. Download the Eclipse IDE from the following URL, and install it according to the instructions: `https://eclipse.org/downloads`
- **Java Card Eclipse Plug-in (optional)** - If you want to use the Java Card Eclipse Plug-in for development and debugging, refer to the Set up the Eclipse Java Card Plug-in section.
- **PCSC-Lite and IFD Handler (optional)** - On a Linux system you could use PCSC-Lite in combination with our `IFDHandler` (*$JC_HOME_SIMULATOR*/`drivers/IFDHandler/libjcsdkifdh.so`) to send APDUs over PC/SC. Please refer to Communication Options Between Host and Java Card Simulator section and the Install and Setup the PCSC-Lite and IFD Handler section for additional instructions.

Build and Install the OpenSSL Library

OpenSSL is a software library that provides cryptographic functionality upon which the Java Card Development Kit is built.

This section describes how to build, install, and set up the OpenSSL library. It includes:

- How to get Open SSL sources
- Build and Configure OpenSSL on Linux
- Build and Configure OpenSSL on Windows

How to Get the OpenSSL Sources

The following series of steps is one of many ways to obtain the OpenSSL Sources.

For additional ways to get the OpenSSL sources, refer to the OpenSSL website: https://www.openssl.org/source/

1. Clone the git repository containing the OpenSSL sources to the current directory.

```
git clone http://www.github.com/openssl/openssl
```

2. List the available versions using the `git tag | grep 3.0` command. You should receive something like this:

```
[…]
openssl-3.0.11
openssl-3.0.12
openssl-3.0.13
[…]
```

3. Select or check out the desired version (for example, 3.0.12).

```
git checkout openssl-3.0.12
```

Build and Configure OpenSSL on Linux

The following series of steps is one way of many to build the OpenSSL sources in Linux.

Example of Building OpenSSL

Refer to the OpenSSL website for additional ways to build with Linux.

1. Update and install build-essential packages. Check for any updates available by updating the package lists from the repository:

```
sudo apt-get update
sudo apt install build-essential checkinstall zlib1g-dev -y
```

2. Install additional 32-bit libraries. For buildling 32-bit ELF format, additional `gcc-multilib` and `g++-multilib` packages must be installed:

```
sudo apt-get install gcc-multilib g++-multilib
```

3. Configure:
   - OpenSSL for 32 bits
   - OpenSSL to be installed in /usr/local/ssl directory
   - No multithreading
   - Disabled assembly language
   - Add support for weak chipers:

   ```
   perl ./Configure linux-x86 no-asm no-threads enable-weak-ssl-ciphers
   --prefix=/usr/local/ssl --openssldir=/usr/local/ssl
   ```

   You should see something similar to this:

   ```
   **********************************************************************
   ***
   ******   OpenSSL has been successfully configured
   ******
   ******   If you encounter a problem while building, please open an
   ******   issue on GitHub and include the output from the following
   ```

```
command:
******

******        perl configdata.pm --
dump
******

******    (If you are new to OpenSSL, you might want to consult
the
******    'Troubleshooting' section in the INSTALL.md file
first)
******

*****************************************************************
********
```

4. Run the build.

```
make
```

5. (Optional) - Test results.

```
make test
```

6. Install.

```
make install
```

Configuring OpenSSL for the Simulator

For the Simulator to access the OpenSSL-3 libraries, follow these steps:

- Set up the *LD_LIBRARY_PATH* environment variable to refer to the directory containig the OpenSSL shared libraries
  For example, if the newly built library is in your `/usr/local/ssl/lib`, do the following:

  ```
  export LD_LIBRARY_PATH=/usr/local/ssl/lib
  ```

  Also, make sure that this directory contains a symbolic link pointing to this version of the library. Otherwise, you an create it by:

  ```
  ln -s /usr/local/ssl/lib/libcrypto.so.3 /usr/local/ssl/lib/
  libcrypto.so
  ln -s /usr/local/ssl/lib/libssl.so.3 /usr/local/ssl/lib/libssl.so
  ```

- Use the legacy provider (`legacy.so`) for "weak SSL ciphers" (like "single DES"), by setting up the *OPENSSL_MODULES* environment variable. For example, if the `legacy.so` library is in your `/usr/local/ssl/lib` folder, do the following:

  ```
  export OPENSSL_MODULES=/usr/local/ssl/lib
  ```

Build and Configure OpenSSL on Windows

The following series of steps could be a way to build OpenSSL sources in Windows.

Example of Building OpenSSL

Refer to the OpenSSL website for more details or other ways to build.

- **Prerequisites**
  Make sure that you have installed all necessary tools as mentioned on the OpenSSL web page. Additionally, ensure that all these tools are accessible via your `%PATH%` setting. Here you will need:
  - `Perl` support (needed for configuration stage)
  - A suitable C compiler suite of your choice (in this example we use the 32-bit version of “`mingw`”)
  - A “`make`” derivate compatible to the C compiler suite of your choice
- **Configure:**
  - OpenSSL for 32 bits (this is done implicitly by the compiler you have selected)
  - No multithreading (option: “no-threads”)
  - Disabled assembly language (option: “no-asm”)
  - Add support for weak ciphers (option: “enable-weak-ssl-ciphers”, this is important if you plan use the simulator with e. g. “single DES” algorithm)

    ```
    perl Configure mingw no-asm no-threads enable-weak-ssl-ciphers
    ```

    You should see something like this at the end:

    ```
    **********************************************************************
    ***                                                                ***
    ***   OpenSSL has been successfully configured                     ***
    ***                                                                ***
    ***   If you encounter a problem while building, please open an    ***
    ***   issue on GitHub                                              ***
    ***   and include the output from the following command:           ***
    ***                                                                ***
    ***       perl configdata.pm --dump                                ***
    ***                                                                ***
    ***   (If you are new to OpenSSL, you might want to consult the    ***
    ***   'Troubleshooting' section in the INSTALL.md file first)      ***
    ***                                                                ***
    **********************************************************************
    ```
- **Run the build.**

  ```
  make
  ```

- **(Optional) - Test results.**

  ```
  make test
  ```

Configuring OpenSSL for the Simulator

- The newly generated DLL files, namely: `libcrypto-3.dll` and `libssl-3.dll` are located in your local build directory. If you want to use the OpenSSL for Java Card Simulator only, copy these files into folder %JC_HOME_SIMULATOR% `\runtime\bin`. If you however want to provide system-wide access, you can copy them into system folder %SystemRoot%`\SysWow64` (in this case you can skip the step mentioned above).
- Use the Legacy provider for "weak SSL ciphers" by copying the additionally generated library file, `legacy.dll`, into the %JC_HOME_SIMULATOR% `\runtime\bin` folder as well.

Install and Setup the PCSC-Lite and IFD Handler

The Java Card Development Kit Simulator works with smart card terminals in Linux operating system by using the PCSC-Lite and the IFD Handler which comes with the bundle.

These are the steps to install, setup, and configure the PC/SC Daemon known as `pcscd` and the IFD Handler library named `libjcsdkifdh.so` located in the *`JC_HOME_SIMULATOR`*`/drivers/IFDHandler/` directory, and how to run it with the simulator.

1. Update `apt` database.

   ```
   sudo apt update
   ```

2. Install middleware to access a smart card using PC/SC Daemon.

   ```
   sudo apt install -y pcscd
   ```

3. Install `pcsc-tools`.

   ```
   sudo apt install -y pcsc-tools
   ```

4. Create a new file named `jcsdk_config` in the PC/SC Daemon default configuration directory named `/etc/reader.conf.d`.

   ```
   sudo touch /etc/reader.conf.d/jcsdk_config
   ```

5. Add a reader configuration for the IFDHandler implementation library named `libjcsdkifdh.so`, in the above-mentioned configuration file, `jcsdk_config`, as described below.

   ```
   # Configuration to interact with the Oracle PCSC Reader for Linux
   FRIENDLYNAME   "Oracle JCSDK PCSC Reader Demo 1"
   DEVICENAME     127.0.0.1:9025
   LIBPATH        ${JC_HOME_SIMULATOR}/drivers/IFDHandler/
   libjcsdkifdh.so
   ```

6. Run PC/SC Daemon with the new configuration for the Java Card Development Kit Simulator IFD Handler.
   Use the following command to start a new daemon on the foreground for debugging.

   ```
   sudo pcscd -f -d -a
   ```

   Or, use the following to restart the pcsc daemon if it is already running. It will refresh the configuration of the reader.

   ```
   sudo systemctl restart pcscd
   ```

   To verify that the `pcscd` configuration is reflected in the new daemon, run `pcsc_scan` to list the available readers. "Oracle JCSDK PCSC Reader Demo 1" should be in the list.
7. Run the simulator for Linux.
   Note: *LD_LIBRARY_PATH* must be updated with the OpenSSL libraries.

   ```
   cd ${JC_HOME_SIMULATOR}/bin
   ./jcsl
   ```

   (Optional) Launch the simulator with desired debugging level.

   ```
   ./jcsl -log_level=finest
   ```

8. Use any client for sending PC/SC commands to the Simulator.

Installing the Java Card Development Kit Simulator

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

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

1. Download the Java Card Development Kit Simulator archive file to a directory of your choice. It is available in the `.zip` file for Windows and `.tar.gz` for Linux.
   - `java_card_devkit_simulator-win-bin-v[version]-buildID-dd-mmm-yyy.zip`
   - `java_card_devkit_simulator-linux-bin-v[version]-buildID-dd-mmm-yyy.tar.gz`
2. Extract the downloaded archive file to the directory of your choice.

   Note:

   The installation directory of the Java Card Development Kit Simulator is referred to as *JC_HOME_SIMULATOR* throughout this documentation.

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

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

Installed Files and Directories

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

Verifying the 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* environment variable must be set to the JDK software root directory and its `bin` folder must be added to the `PATH`.
- *JC_HOME_SIMULATOR* environment variable must be set to the Java Card Development Kit Simulator root directory.
- *JC_HOME_TOOLS* environment variable must be set to the Java Card Development Kit Tools root directory.

Configuring the Java Card Development Kit Simulator

Before using the simulator for the first time, it must be provisioned with an initial Secure Channel Protocol key set and with a Global PIN.

This can be done by running the Configurator tool from the command line as follows:

```
java -jar Configurator.jar -binary <simulator-binary> -SCP-keyset
<k_enc> <k_mac> <k_dek> -global-pin <pin> <tries_count>
```

where:

- **simulator-binary** is the SDK binary file to inject the data, which is `jcsl` for Linux and `jcsw.exe` for Windows.
- **k_enc** is the ENC key of the card manager SCP keyset. It is not set by default.
- **k_mac** is the MAC key of the card manager SCP keyset. It is not set by default.
- **k_dek** is the DEK key of the card manager SCP keyset. It is not set by default.
- **pin** is the global Personal Identification Number (PIN) used as a mechanism for CVM.
- **tries_count** is the CVM Retry Limit.

> **Note:**
>
> The `Configurator.jar` file is located in the `tools` sub-directory of *JC_HOME_SIMULATOR*. To print the available arguments and further details of the configurator, execute `java -jar Configurator.jar`

Here are examples on how to provision the simulator using the Configurator. Use your own values accordingly.

Example - Linux (values must be set with yours):

```
java -jar ${JC_HOME_SIMULATOR}/tools/Configurator.jar -binary $
{JC_HOME_SIMULATOR}/runtime/bin/jcsl -SCP-keyset
1111111111111111111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222222222222222222
3333333333333333333333333333333333333333333333333333333333333333 -global-pin
01020304050f 03
```

Example - Windows

```
java -jar %JC_HOME_SIMULATOR%\tools\Configurator.jar -binary
%JC_HOME_SIMULATOR%\runtime\bin\jcsw.exe -SCP-keyset
1111111111111111111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222222222222222222
3333333333333333333333333333333333333333333333333333333333333333 -global-pin
01020304050f 03
```

Uninstalling the Java Card Development Kit Simulator

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

1. Start Eclipse and remove the Java Card Plug-in:
   a. From the Eclipse **Help** menu, select **About Eclipse**.
   b. On the **Installed Software** tab, select **Java Card 3 Platform Development Kit**.
   c. Click **Uninstall...** and follow the prompts.
2. Delete the *JC_HOME_SIMULATOR* directory from your hard drive, if required.

Set up the Eclipse Java Card Plug-in

The Java Card Development Kit Eclipse Plug-in assists you in developing Java Card applications. Almost all of the choices presented by the plug-in dialogs correspond to command-line options of the tools in the Java Card Development Kit Tools bundle (`converter` and `verifier`) which are described in the Java Card Development Kit Tools Users 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
- Configuring Sample_Platform and Sample_Device

- Configuring the Java Card Tools Path

Installing the Eclipse Plug-in

The Java Card Development Kit Eclipse Plug-in is available for download at https://www.oracle.com/java/java-card/. The Eclipse Plug-in provides a convenient way to develop Java Card applets. To install the Plug-in:

1. Download the Java Card Development Kit Eclipse Plug-in archive file to a directory of your choice. It is available in a .zip file for both Windows and Linux.
   - java_card_devkit_eclipse_plugin-bin-v[version]-<buildID>-<dd-mmm-yyy>.zip
2. Extract the downloaded archive file to the directory of your choice.
3. In the Eclipse menu bar, select **Help**, and then select **Install New Software**.
4. Click **Add**.
5. Click **Archive**.
6. Select the Eclipse Plug-in zip file.
7. In the Add Repository dialog, type Java Card SDK in the **Name** field.
8. Click **Add**.
9. On the Available Software dialog, select the feature to install:

   Java Card 3 Platform Development Kit

   Note:

   If you don't see any items and the message There are no categorized items appears, then you must uncheck the **Group items by category** checkbox.

10. Click **Next** until the terms of the licenses are displayed.
11. Please read and accept the terms of the licenses and click **Finish** to complete installation.
12. When the software has been installed, Eclipse will prompt you to restart. Click **Yes**.
13. The Eclipse Plug-in has module dependencies. To provide these dependencies to Eclipse, you must edit the eclipse.ini file, which can be found inside the Eclipse installation folder. The following two lines must be added after -vmargs:
    - On Linux:

```
--module-path=${JC_HOME_SIMULATOR}/client/AMService/
amservice.jar:${JC_HOME_SIMULATOR}/client/COMService/
extension.jar:${JC_HOME_SIMULATOR}/client/COMService/
socketprovider.jar
--add-modules=ALL-MODULE-PATH
```

Please be sure that the *LD_LIBRARY_PATH* environment variable is set to the OpenSSL library location before running Eclipse.

- On Windows:

```
--module-path=%JC_HOME_SIMULATOR%
\client\AMService\amservice.jar;%JC_HOME_SIMULATOR%
\client\COMService\extension.jar;%JC_HOME_SIMULATOR%
\client\COMService\socketprovider.jar
--add-modules=ALL-MODULE-PATH
```

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 the following:

- `Sample_Platform`, which points to the directory that is set in the *JC_HOME_SIMULATOR* environment variable, and
- `Sample_Device`, which contains the settings for the Simulator.

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

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

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

Configuring Sample_Platform

You may want to configure a `Sample_Platform`.

These are the steps to configure the `Sample_Platform`:

1. In Eclipse, from the **Window** menu, select **Preferences**.
2. In the Preferences dialog, click on **Java Card Platforms** and select **Sample_Platform**.
3. Press **Edit...** button.
4. In the edit window dialog, set **Java Card SDK path** to *JC_HOME_SIMULATOR*.
5. Press **OK** button.

Configuring Sample_Device

These are the steps to configure the `Sample_Device`:

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** and select **Java Card Devices**.
3. Check that **Sample_Platform** is selected in the upper combo box.
4. Select **Sample_Device.**
5. Press **Edit...** button.
6. In the case of missing `config` file, which is by default missing, you will see an error in the upper part of the dialog. The configuration file must be set in the **Config file:** field.
   The configuration file must contain the following entries:

```
ISD=<IssuerSecurityDomainAID>
<IssuerSecurityDomainAID>_scp03enc_10=<k_enc>
<IssuerSecurityDomainAID>_scp03mac_10=<k_mac>
<IssuerSecurityDomainAID>_scp03dek_10=<k_dek>
```

   For example, the following entries match the information in the example provided in Configuring the Java Card Development Kit Simulator when the simulator is provisioned for the first time.

```
ISD=A000000151000000
A000000151000000_scp03enc_10=111111111111111111111111111111111111111
1111111111111111111111111
A000000151000000_scp03mac_10=222222222222222222222222222222222222222
2222222222222222222222222
A000000151000000_scp03dek_10=333333333333333333333333333333333333333
3333333333333333333333333
```

7. Press **OK** button.

Configuring the Java Card Tools Path

To enable the Eclipse Plug-in to build a Java Card project, you must configure the Java Card Tools path. This is the path where the tools bundle is present.

The tools bundle includes the `converter` and `verifier` tools. This is a workspace-wide setting. Therefore, the tools bundle is used for all Java Card projects that are created in the same workspace. To change the Java Card Tools path for the workspace:

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

3

Running the Simulator and the Samples

A number of example programs are provided with the development kit. All of these samples, showing up basic functionality, are located under *JC_HOME_SIMULATOR*`\samples`.

This chapter contains the following sections:

- Java Card Development Kit Simulator Command Line
- How to Run the Samples
- Running the Included Samples

Java Card Development Kit Simulator Command Line

The Simulator supports the following command line parameters.

Table 3-1 Command Line Parameters

Parameter	Description
`-h` or `-help`	Prints help screen showing all available command line options.
`-log_level=<level>`	Selects the level of details provided in the log. Log detail levels are: `finest`, `finer`, `fine`, `config`, `info` (default), `warning`, `severe`
`-i=<input file>`	Specifies the input file to load a binary image into Non-Volatile Memory.
`-o=<output file>`	Specifies the output file to store the binary image for Non-Volatile Memory.
`-iin=<hexstring>`	Initializes the Issuer Identification Number (8 byte max).
`-cin=<hexstring>`	Initializes the Card Image Number (10 bytes max).
`-lcEncoding=<type>`	Sets the APDU Logical Channel encoding: `type4` or `type4type16` (default).
`-p=<number>`	Starts the Simulator on the specified port number (default is 9025).
`-debug_port=<number>`	Starts the debugger on the specified port number.

For example, on Linux:

```
$ ./jcsl -p=9000 -log_level=finest
INFO    |msg|000001|grp:hal | Oracle Java Card Simulator (v24.0 - Jan 14 2024)
INFO    |msg|000002|grp:hal |  - Java Card v3.2
INFO    |msg|000003|grp:hal |  - GP Card v2.3
```

For example, on Windows:

```
>jcsw.exe -p=9000 -log_level=finest
INFO   |msg|000001|grp:hal | Oracle Java Card Simulator (v24.0 - Jan
14 2024)
INFO   |msg|000002|grp:hal |  - Java Card v3.2
INFO   |msg|000003|grp:hal |  - GP Card v2.3
```

How to Run the Samples

Note that before you can run the samples, the file *JC_HOME_SIMULATOR*/samples/client.config.properties needs to be configured accordingly. For example, lines like these:

```
A000000151000000_scp03enc_10=<Enter your 32-bit enc key here>
A000000151000000_scp03mac_10=<Enter your 32-bit mac key here>
A000000151000000_scp03dek_10=<Enter your 32-bit dek key here>
```

must be changed to something like this:

```
A000000151000000_scp03enc_10=111111111111111111111111111111111111111111
1111111111111111111111
A000000151000000_scp03mac_10=222222222222222222222222222222222222222222
2222222222222222222222
A000000151000000_scp03dek_10=333333333333333333333333333333333333333333
3333333333333333333333
```

For additional information on configuring these properties, refer to Configuring the Java Card Development Kit Simulator.

Each sample directory contains an `applet` folder and a `client` folder. You can use the Eclipse Plug-in or the script files (`build.bat` / `run.bat` for Windows, `build.sh` / `run.sh` for Linux) to be used 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 and verify the class files.

The Java Card runtime environment, `jcsw.exe` (on Windows), `jcsl` (on Linux) respectively, simulates a Java Card 3.2 platform on a smart card. Applets are installed into the runtime environment, and it simulates interaction with a card reader.

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, import the projects, build them, start the device, and then run the sample-specific `client` Java application which loads the applet(s), sends necessary APDUs, and cleans up afterwards.

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 (`converter` and `verifier`) which are described in Running the Samples from the Command Line. The plug-in runs those tools with the options that you select.

Here are a few notes on running the samples.

Sample_Platform and Sample_Device

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

Java Card View

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

Importing and Building Projects

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

After you have imported 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

The output files created by the converter (`cap`, `jca`, and `exp` files) are put in the the `deliverables` directory.

Running Sample_Device

After having built the Java Card applet(s) the according `client` project has to be imported. To do so, use the **File** menu, select **Import > General > Projects from Folder or Archive** to import the according Java `client` project. Make sure that you select the directory that has Java source files in it from the project. In most cases, this directory is the `client` folder.

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

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

```
help
```

You should see a screen with all options available.

Sample_Device Settings

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

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

Classpath Variable Settings

All samples come with a pre-defined classpath file including all necessary external libraries. These classpath files are part of the according Eclipse projects and use an internally defined classpath variable to automatically match the installation folder.

To create or modify a classpath variable select **Window > Preferences**. Expand **Java > Build Path > Classpath variables** and check for *JC_HOME_SIMULATOR*. If it does not already exist, create it and make sure to let it point to the root directory of the Simulator bundle installed on your machine.

External JREs vs. Eclipse internal JREs

We recommend that you install either Oracle Java SE 17 or OpenJDK Version 17, as these were the versions that were tested. After installation, select **Window > Preferences**. Expand **Java > Installed JREs**, click "Add" and, select the installed JDK. Make sure that you have checked the corresponding check box to ensure it acts as the default JRE.

To check this, select **Window > Preferences**. Expand **Java > Installed JREs > Execution Environment** and check an external version of your choice. The line will change to bold style and a comment “(default)”, also in bold style will be added.

Figure 3-1 Execution Environments

Running the Samples from the Command Line

To build and run the samples:

1. In a Command Prompt window navigate to the sample you want to evaluate, build all necessary modules by entering either `build.bat` or `./build.sh` depending on your operating system. **Note**, all the samples are located under: *JC_HOME_SIMULATOR*`\samples`.
2. Start-up simulator by running either (also depending on your operating system)
   - `%`*`JC_HOME_SIMULATOR`*`%\runtime\bin\jcsw.exe`
   - `${`*`JC_HOME_SIMULATOR`*`}/runtime/bin/jcsl`
3. In the Command Prompt window created above execute the sample's client by running `run.bat` or `./run.sh` respectively.

Running the Included Samples

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

- HelloWorld Sample - A minimal applet utilizing the simplest source code and meta-files that demonstrates the base structure of a Java Card applet that developers can use to develop, deploy, create, execute, delete, and unload a standalone module.
- Channels Sample - Demonstrates the use of logical channels which allows selecting multiple applets at the same time.
- 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 - Demonstrates applet and package deletion and the object deletion mechanism that removes unreachable objects.
- SignatureMessageRecovery Sample - Demonstrates message recovery. This sample is only included in bundles with cryptography extensions.
- ArrayViews Sample - Demonstrates a client/server application sharing data using array views.

HelloWorld Sample

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

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

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

Running the HelloWorld Sample in Eclipse

Run the `HelloWorld` sample using the Java `client` application.

Start Eclipse. `Sample_Platform` and `Sample_Device` must have already been created.

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

   The build puts the ouput from the converter (`cap`, `jca`, and `exp` files) in the `deliverables` directory.

2. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `client` directory from the HelloWorld project, to import the HelloWorld Java client into your workspace.
3. If you do not see the Java Card view, go to the **Window** menu, select **Show View >Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.
4. In the Java Card View, right-click on **Sample_Device** and select **Start**.

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

5. Using the **File** menu, select **Import > Run/Debug > Launch Configurations**, select the HelloWorld sample directory and check file `HelloWorld.launch` to import the HelloWorld Java client launch configuration into Eclipse.
6. Using the **Run** menu, select **Run Configurations...** or **Debug Configurations...** to run or debug the imported `HelloWorld` configuration

Running the HelloWorld Sample from the Command Line

To run the `HelloWorld` sample:

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

      `jcsw` (on Windows systems) or `./jcsl` (on Linux systems).
2. Open a second Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\HelloWorld` directory.
   b. Enter either `build.bat` or `./build.sh` (according to your operating system) in order to build Java Card applet and Java client.
   c. Enter either `run.bat` or `./run.sh` (according to your operating system) to load the applet into the Simulator, send all APDUs and clean up afterwards.

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 `Channels` sample also demonstrates the use of `ExtendedLength` APDU.

The `Channels` sample mimics the behavior of a 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.

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. `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 deleted

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

Run the `Channels` sample using the Java `client` application.

Start Eclipse. `Sample_Platform` and `Sample_Device` must have already been created.

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

   The build puts the output from the converter (`cap`, `jca`, and `exp`) files in the deliverables directory.
2. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `client` directory from the Channels project, to import the Channels Java client into your workspace.
3. If you do not see the Java Card view, go to the **Window** menu, select **ShowView > Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.
4. In the Java Card View, right-click on **Sample_Device** and select **Start**. The simulator starts and you can see the output in the Console view.
5. Using the **File** menu, select **Import > Run/Debug > Launch Configurations**, select the Channels sample directory and check the file `Channels.launch` to import the Channels Java client launch configuration into Eclipse.
6. Using the **Run** menu, select **Run Configurations …** or **Debug Configurations ...** to run or debug the imported `Channels` configuration.

Running the Channels Sample from the Command Line

To run the `Channels` sample:

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

      `jcsw` (on Windows systems) or `./jcsl` (on Linux systems).
2. Open a second Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\Channels` directory.

b. Enter either `build.bat` or `./build.sh` (according to your operating system) in order to build Java Card applets and Java client.

c. Enter either `run.bat` or `./run.sh` (according to your operating system) to load the applet into the Simulator, send all APDUs and clean up afterwards.

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 for 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

Run the `Utility` sample using the Java `client` application.

Start Eclipse. `Sample_Platform` and `Sample_Device` must have already been created.

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

   The build puts the output from the converter (`cap`, `jca`, and `exp` files) in the `deliverables` directory.

2. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `client` directory from the Utility project, to import the Utility Java client into your workspace.
3. If you do not see the Java Card view, go to the Window menu, select **ShowView > Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.
4. In the Java Card View, right-click on **Sample_Device** and select **Start**. The Simulator starts and you can see the output in the Console view.
5. Using the **File** menu, select **Import > Run/Debug > Launch Configurations**, select the Utility sample directory and check file `Utility.launch` to import the Utility Java client launch configuration into Eclipse.
6. Using **Run** menu, select **Run Configurations …** or **Debug Configurations ...** to run or debug the imported `Utility` configuration.

Running the Utility Sample from the Command Line

To run the `Utility` sample:

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

   a. Navigate to the `JC_HOME_SIMULATOR\runtime\bin` directory.
   b. Start the Simulator by entering one of the following commands at the command prompt:

      `jcsw` (on Windows systems) or `./jcsl` (on Linux).

2. Open a second Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\Utility` directory.
   b. Enter either `build.bat` or `./build.sh` (according to your operating system) in order to build Java Card applet and Java client.
   c. Enter either `run.bat` or `./run.sh` (according to your operating system) to load the applet into the Simulator, send all APDUs and clean up afterwards.

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.

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

Run the `Wallet` sample using the Java `client` application.

Start Eclipse. `Sample_Platform` and `Sample_Device` must have already been created.

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

   The build puts the output from the converter (`cap`, `jca`, and `exp` files) in the `deliverables` directory.

2. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `client` directory from the Wallet project, to import the Wallet Java client into your workspace.
3. If you do not see the Java Card view, go to the **Window** menu, select **ShowView > Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.
4. In the Java Card View, right-click on **Sample_Device** and select **Start**. The Simulator starts and you can see the output in the Console view.
5. Using the **File** menu, select **Import > Run/Debug > Launch Configurations**, select the Wallet sample directory and check file `Wallet.launch` to import the Wallet Java client launch configuration into Eclipse.
6. Using the **Run** menu, select **Run Configurations …** or **Debug Configurations ...** to run or debug the imported `Wallet` configuration.

Running the Wallet Sample from the Command Line

To run the `Wallet` sample:

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

      `jcsw` (on Windows systems) or `./jcsl` (on Linux systems)
2. Open a second Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\Wallet` directory.
   b. Enter either `build.bat` or `./build.sh` (according to your operating system) in order to build Java Card applet and Java client.
   c. Enter either `run.bat` or `./run.sh` (according to your operating system) to load the applet into the Simulator, send all APDUs and clean up afterwards.

ObjectDeletion Sample

The `ObjectDeletion` sample demonstrates the object deletion mechanism, applet deletion and package deletion:

- “Stage preparation”: 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.
- “Stage exec_1_1”: 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.
- “Stage exec_1_2”: 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.

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

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

Running the ObjectDeletion Sample in Eclipse

Run the `ObjectDeletion` sample using the Java `client` application.

Start Eclipse. `Sample_Platform` and `Sample_Device` must have already been created.

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

The build puts the output from the converter (`cap`, `jca`, and `exp` files) in the `deliverables` directory.

2. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `client` directory from the ObjectDeletion project, to import the `ObjectDeletion` Java client into your workspace.
3. If you do not see the Java Card view, go to the **Window** menu, select **ShowView > Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.
4. In the **Java Card View**, right-click on **Sample_Device** and select **Start**. The Simulator starts and you can see the output in the **Console** view.
5. Using the **File** menu, select **Import > Run/Debug > Launch Configurations**, select the ObjectDeletion sample directory and check the file `ObjectDeletion.launch` to import the ObjectDeletion Java client launch configuration into Eclipse.
6. Using the **Run** menu, select **Run Configurations …** or **Debug Configurations ...** to run or debug the imported `ObjectDeletion` configuration.

Running the ObjectDeletion Sample from the Command Line

To run the `ObjectDeletion` sample using command line (on Linux systems make sure that executable flags for executable files are set correctly):

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

      `jcsw` (on Windows systems) or `./jcsl` (on Linux systems).
2. Open a second Command Prompt window, perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\ObjectDeletion` directory.
   b. Enter either `build.bat` or `./build.sh` command at the command prompt (according to your operating system) in order to build Java Card applets and Java client.
   c. Enter either `run.bat` or `./run.sh` (according to your operating system) to load the applets into the Simulator, send all APDUs and clean up afterwards.
3. Inspect client’s output on the console – all APDUs should have been confirmed by the simulator with a positive response APDU (0x9000).

SignatureMessageRecovery Sample

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 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 the sample application a large message to sign.

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.

The second script 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 this script are:

1. The script sends the sample application a small message to be signed.
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.

Follow one of these set of instructions to run the sample:

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

Running the SignatureMessageRecovery Sample in Eclipse

Run the `SignatureMessageRecovery` sample using the Java `client` application.

Start Eclipse. `Sample_Platform` and `Sample_Device` must have already been created.

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

   The build puts the output from the converter (`cap`, `jca`, and `exp` files) in the `deliverables` directory.
2. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `client` directory from the `SignatureMessageRecovery` project, to import the `SignatureMessageRecovery` Java client into your workspace
3. If you do not see the **Java Card view**, go to the **Window** menu, select **ShowView > Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.

4. In the **Java Card view**, right-click on **Sample_Device** and select **Start**. The Simulator starts and you can see the output in the **Console** view.
5. Using the **File** menu, select **Import > Run/Debug > Launch Configurations**, select the SignatureMessageRecovery sample directory and check file `SignatureMesssageRecovery.launch` to import the SignatureMessageRecovery Java client launch configuration into Eclipse.
6. Using **Run** menu, select **Run Configurations …** or **Debug Configurations ...** to run or debug the imported `SignatureMessageRecovery` configuration.

Running the SignatureMessageRecovery Sample from the Command Line

To run the `SignatureMessageRecovery` sample:

1. Open a Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\runtime\bin` directory.
   b. Start the Simulator by entering one of the following commands at the command prompt:
      `jcsw` (on Windows systems) or `./jcsl` (on Linux systems).
2. Open a second Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\SignatureMessageRecovery` directory.
   b. Enter either `build.bat` or `./build.sh` (according to your operating system) in order to build Java Card applet and Java client.
   c. Enter either `run.bat` or `./run.sh` (according to your operating system) to load the applet into the Simulator, send all APDUs and clean up afterwards.

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.

ArrayViews Sample

The `ArrayViews` sample comes with the following three source files:

- Applet `ClientApplet`
- Library package `MyShareable`
- Applet `ServerApplet`

Several techniques are demonstrated here:

1. A client - server application (two separate applets) shares data using array views: Using this allows applications to access a certain part of an array as if it was a standalone array
2. Usage of `ShareableInterface` for communication purposes
3. Resource file contents using `javacard.framework.Resources`

Note:

The converter generates the application files one by one where the package `MyShareable` must be the first one to be converted, as it is needed for the other two applications as well.

This section contains:

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

Running the ArrayViews Sample in Eclipse

Run the `ArrayViews` sample using the Java `client` application.

Start Eclipse. `Sample_Platform` and `Sample_Device` must have already been created.

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

   The build puts the output from the converter (`cap`, `jca`, and `exp`) files in the `deliverables` directory.

2. Using the **File** menu, select **Import > General > Projects from Folder or Archive**, and select the `client` directory from the ArrayViews project, to import the ArrayView Java client into your workspace.
3. If you do not see the **Java Card view**, go to the **Window** menu, select **ShowView > Other...** In the list, expand **Oracle Java Card SDK** and select **Java Card view**.
4. In the **Java Card view**, right-click on **Sample_Device** and select **Start**. The Simulator starts and you can see the output in the **Console** view.

5. Using the **File** menu, select **Import > Run/Debug > Launch Configurations**, select the ArrayViews sample directory and check the file `ArrayViews.launch` to import the ArrayViews Java client launch configuration into Eclipse.
6. Using the **Run** menu, select **Run Configurations …** or **Debug Configurations ...** to run or debug the imported `ArrayViews` configuration.

Running the ArrayViews Sample from the Command Line

To run the `ArrayViews` sample using command line (on Linux systems make sure that executable flags for executable files are set correctly):

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

      `jcsw` (on Windows systems) or `./jcsl` (on Linux systems)
2. Open a second Command Prompt window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR\samples\ArrayViews` directory.
   b. Enter either `build.bat` or `./build.sh` (according to your operating system) in order to build Java Card applets and Java client.
   c. Enter either `run.bat` or `./run.sh` (according to your operating system) to load the applets into the Simulator, send all APDUs and clean up afterwards.
3. Inspect the client's output on the console – all APDUs should have been confirmed by the simulator with a positive response APDU (0x9000).

4

Debugging Applications

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

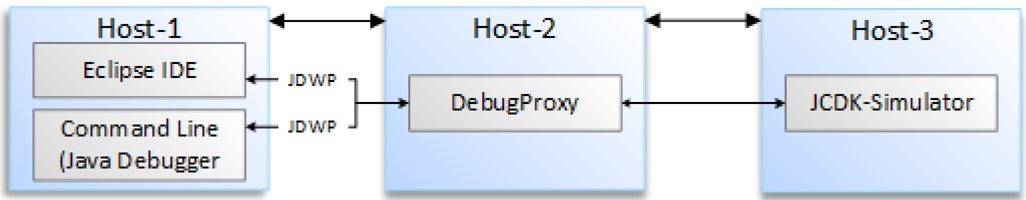
- Debugger Architecture
- Running the Debug Proxy from the Command Line

Debugger Architecture

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

The prebuilt executable runtime environment, `JCDK-Simulator`, is run from inside the JCDK Eclipse Plug-in or on the command line. It performs I/O through a socket interface, simulating the interaction between a card reader and a host computer.

Figure 4-1 Debugging Architecture



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

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

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

Running the Debug Proxy from the Command Line

If you are not using the JCDK Eclipse Plug-in 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. Use the `AMService` API to write a client for applet management.
3. Start the `JCDK-Simulator` in debug mode.

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

   For example (depending on your operating system):

   %*JC_HOME_SIMULATOR*%\runtime\bin\jcsw.exe -debug_port=9090 [options]

   ${*JC_HOME_SIMULATOR*}/runtime/bin/jcsl -debug_port=9090 [options]

4. Run the Java client to deploy the applet.

   At a minimum, the applet load must be executed before the debug proxy connects to the VM. Other commands can be executed later to debug the applet's `install()` and `process()` methods.

5. Start `jc-debug-proxy`.

   For example:

   ```
   java -jar jc-debug-proxy.jar -capPath
   ${JC_HOME_SIMULATOR}/samples/HelloWorld/applet/deliverables/
   HelloWorld/com/oracle/jcclassic/samples/helloworld/javacard/
   helloworld.cap -port 8000 -vmPort 9090
   ```

6. Attach the debugger to the debug proxy.

   A 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 -jar jc-debug-proxy.jar <debug proxy arguments>
```

The command line arguments for the debug proxy are:

Command Line Argument	Description
`-help`	Short description of help.
`-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. Default value: 8000.
`-vmPort`	The port that the VM listens on. Default value: 9090.
`-vmHost`	The hostname of the system the VM is running on. Default value: localhost.

Command Line Argument	Description
-capPath	Required. Path to the cap file(s) being debugged.

For example:

```
java -jar jc-debug-proxy.jar -capPath ${JC_HOME_SIMULATOR}/samples/
HelloWorld/applet/deliverables/HelloWorld/com/oracle/jcclassic/samples/
helloworld/javacard/helloworld.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 terminal window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR/runtime/bin` directory.
   b. Start the simulator by entering the following command at the command prompt:
      `[jcsw.exe|./jcsl] -debug_port=9090`
2. Open a second terminal window and perform the following:
   a. Navigate to the `JC_HOME_SIMULATOR/samples/HelloWorld` directory.
   b. Edit `AMSHelloWorldClient.java` to send only the deploy script.
   c. Execute the build script:
      `[build.bat|./build.sh]`
      This builds the applet and the client.
   d. Execute the run script:
      `[run.bat|./run.sh]`
      This will deploy the applet.
3. In the second command prompt, navigate to the `JC_HOME_SIMULATOR/client/DebugProxy` directory and start the debug proxy:

```
java -jar jc-debug-proxy.jar -capPath ${JC_HOME_SIMULATOR}/samples/
HelloWorld/applet/deliverables/HelloWorld/com/oracle/jcclassic/samples/
helloworld/javacard/helloworld.cap
```

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

5

Packaging and Deploying Your Application

This chapter describes how to prepare your application (applet) to be put into a CAP file so it can be deployed to a secure element or to the JDK Simulator.
Applet installation is performed based on GlobalPlatform [GP] SCP03 commands, please refer to Annex B for the details of the GP implementation supported by the JCDK Simulator.
For every sample provided with JCDK there exists a subdirectory (`client`) containing a Java client application (using `AMService`) that shows how to deploy and delete packages and applets.

This chapter contains the following sections:

- Overview of Packaging and Deploying Applications
- Installer Components and Data Flow
- Running Client Application for Applet Management
- Downloading CAP Files and Creating Applets
- Using the On-card Installer for Deletion

Overview of Packaging and Deploying Applications

You can:

- Download a Java Card technology CAP file to the Java Card Simulator
- Instantiate a Java Card applet and perform necessary on-card linking
- Delete applets and packages from the Java Card Simulator

The GP Issuer Security Domain (ISD) is not a multi-selectable application. On startup, the ISD is the default applet on logical channel 0. The default applet on the other logical channels is set to No applet selected.

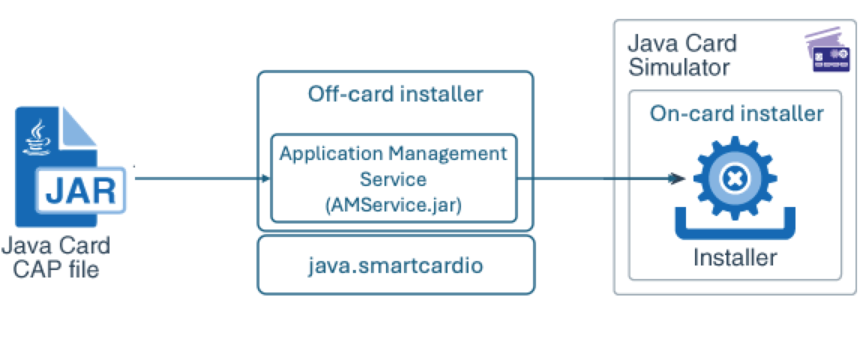
Installer Components and Data Flow

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

The off-card installer is any application that is Global Platform (GP) compliant and supports SCP 03. The following description displays the usage of the `AMService` API from the stand-alone Java client applcation. In the case of the JCDK simulator, the on-card installer is also a GP ISD.

For more information about the installer, see the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.2*.

Figure 5-1 Installed Components Data Flow



The data flow of the installation process is as follows:

1. The GP SCP03 client application takes the CAP file produced by the Converter and send the CAP file as APDU commands to the GP ISD.
2. The GP ISD installer processes the CAP file contents enveloped in the APDU commands, sends a response APDU containing a status and, optionally, the response data.

Running Client Application for Applet Management

This is a guidance for compiling and running with command line parameters in Linux and Windows.

Please be sure that the simulator (`[${JC_HOME_SIMULATOR}/runtime/bin/jcsl | %JC_HOME_SIMULATOR%\runtime\bin\jcsw.exe]`) is started properly (e.g. in Linux the `LD_LIBRARY_PATH` environment variable must be properly exported).

Use the following commands to compile and run the Java client application:

```
[export | set] JAVA_HOME=<<JAVA_DEVKIT_PATH>>                # path to
OpenJDK Java Standard Edition
[export | set] JC_HOME_TOOLS=<<JAVACARD_TOOLS_PATH>>         # path to
the latest Java Card Tools
[export | set] JC_HOME_SIMULATOR=<<JAVACARD_SIMULATOR_PATH>> # path to
Java Card Simulator from this SDK

# Compiling Java client on Linux:
export EXT_MODULEPATH=$JC_HOME_SIMULATOR/client/
AMService:$JC_HOME_SIMULATOR/client/COMService # path to the modules
of the Client API Application Management Service
$JAVA_HOME/bin/javac -cp
        .:$JC_HOME_SIMULATOR/client/AMService/
amservice.jar:$JC_HOME_SIMULATOR/client/COMService/
socketprovider.jar:$JC_HOME_SIMULATOR/client/COMService/
extension.jar
        --module-path $EXT_MODULEPATH --add-modules ALL-MODULE-PATH
<<Java client source file(s)>>
```

```
# Compiling Java client on Windows:
set EXT_MODULEPATH=%JC_HOME_SIMULATOR%\client\AMService;%JC_HOME_SIMULATOR%
\client\COMService # path to the modules of the Client API Application
Management Service
%JAVA_HOME%\bin\javac -cp
        .;%JC_HOME_SIMULATOR%
\client\AMService\amservice.jar;%JC_HOME_SIMULATOR%
\client\COMService\socketprovider.jar;%JC_HOME_SIMULATOR%
\client\COMService\extension.jar
        --module-path %EXT_MODULEPATH% --add-modules ALL-MODULE-PATH <<Java
client source file(s)>>

# Running Java client on Linux:
$JAVA_HOME/bin/java -cp
        .:$JC_HOME_SIMULATOR/client/AMService/
amservice.jar:$JC_HOME_SIMULATOR/client/COMService/
socketprovider.jar:$JC_HOME_SIMULATOR/client/COMService/extension.jar
        --module-path $EXT_MODULEPATH --add-modules ALL-MODULE-PATH <<Java
client class file>> <<parameter(s)>>

# Running Java client on Windows:
%JAVA_HOME%\bin\java -cp
        .;%JC_HOME_SIMULATOR%
\client\AMService\amservice.jar;%JC_HOME_SIMULATOR%
\client\COMService\socketprovider.jar;%JC_HOME_SIMULATOR%
\client\COMService\extension.jar
        --module-path %EXT_MODULEPATH% --add-modules ALL-MODULE-PATH <<Java
client class file>> <<parameter(s)>>
```

Downloading CAP Files and Creating Applets

The procedures for CAP file deployment and applet instance creation are described in the following sections, as are the off-card installer API calls. The default GP ISD app is handling all management APDU commands received from the client application. Each secure element / smart card includes a GP Card Manager with its own AID.

On-Card Installer Applet AID

See Annex B for the AID of the GP ISD

Deploying the CAP File and Creating an Applet Instance

The source code for deploying CAP file and creating an applet instance is included in the samples directory of the Simulator bundle. Each sample has a `client` subdirectory, which contains the source code for the client side.

The Simulator's on-card installer sends back APDU responses and status words compliant with Global Platform.

Using the On-card Installer for Deletion

The GS IPD in the simulator provides the ability to delete CAP files and applet instances from the card's memory. Once GS IPD is selected, it can receive deletion requests from a client application (off-card) using `AMService` API which is sending APDUs compliant with GP

SCP03. Requests to delete an applet or CAP file cannot be sent from an applet on the card. For more information on CAP file and applet deletion, see the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.2*.

How to Send a Deletion Request

The source code for sending deletion requests is included in the samples directory of the Simulator bundle. Each sample has a `client` subdirectory, which contains the source code for the client side.

Configuration File

The configuration file used for the client off-card application for the file `config.properties` is described in detail in the Configuring Sample_Device section.

The client `AMService` API needs the configuration file for both deployment and deletion of the applets and packages.

Part II

Programming With the Development Kit

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

- Using Object, CAP File and Applet Deletion
- Working with Logical Channels
- Using Extended APDU
- Java Card Development Kit Accessibility Information

6 Using Object, CAP File and Applet Deletion

This chapter first describes how to use the object deletion mechanism, then show the Java Card Platform functions for deleting CAP files and applets.

This section includes the following topics:

- Object Deletion Mechanism
- CAP File and Applet Deletion

Object Deletion Mechanism

The object deletion mechanism on the Java Card Platform, Version 3.2 reclaims memory that is being used by "unreachable" objects. Objects become unreachable for a number of reasons such as static or instance fields having missing pointers, missing variable references (not only fields), or when the object is orphaned in an island of isolation. An applet object is reachable until it is successfully deleted.

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

Due to these limitations, the object deletion mechanism in Java Card technology is not triggered automatically. It is performed only when an applet requests it.

Note:

Use the object deletion mechanism sparingly and only when other Java Card technology-based facilities are cumbersome or inadequate.

Requesting the Object Deletion Mechanism

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

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

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

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

Object Deletion Mechanism Usage Guidelines

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

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

CAP File and Applet Deletion

In Java Card Platform, Version 3.2, the installer deletes CAP files and applets from the card's memory. Once the installer is selected, it can receive requests from the terminal, in the form of an APDU, to delete CAP files and applets. Requests to delete an applet or CAP file cannot be sent from an applet on the card.

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

- Developing Removable CAP File
- Writing Removable Applets

Developing Removable CAP File

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

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

CAP file deletion will not succeed if one of the following conditions applies:

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

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

Writing Removable Applets

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

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

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

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

The AppletEvent.uninstall Method

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

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

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

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

Applets can be deleted at any time therefore implement the `uninstall` method defensively. Write your applet with these guidelines in mind:

- The applet continues to function consistently and securely if deletion fails
- The applet can withstand a possible tear during the execution
- The `uninstall` method can be called again if deletion is reattempted

The following example shows such an implementation:

```
public class TestApp1 extends Applet implements AppletEvent{
    // field set to true after uninstall
    private boolean isAppDisabled = false;
    ...
    public void uninstall(){
        if (!isAppDisabled){
            JCSystem.beginTransaction();     //to protect against tear
            isAppDisabled = true;            //mark as uninstalled
            TestApp2SIO.removeDependency();
            JCSystem.commitTransaction();
        }
    }
    public boolean select(boolean appInstAlreadyActive) {
        // refuse selection if in uninstalled state
        if (isAppDisabled) return false;
        return true;
    }
    ...
}
```

7 Working with Logical Channels

The Java Card, Version 3.2 platform can support up to twenty logical channels per active interface. Logical channels allow the concurrent execution of multiple applications on the card, allowing a terminal to handle different tasks at the same time.
This chapter includes the following topics:

- Dual Interface Cards
- Applets and Logical Channels
- The MultiSelectable Interface
- Writing Applets for Concurrent Logical Channels

Dual Interface Cards

On dual interface cards, each interface can handle up to twenty independent logical channels. Channel management commands only affect the logical channels in the interface where the commands are issued.

See the *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.2* for more information on logical channels, their implementation, and logical channel terminology.

Applets and Logical Channels

If you design your applets to take advantage of multi-session functionality, they can interoperate from different channels and can be selected multiple times in different channels. For example, the card might handle security information on one channel, while data is accessed on a second channel, while the third channel takes care of data encoding operations.

By following this design, it is possible to access information owned by a different applet without having to deselect the currently selected applet that is handling session information. Thus, you avoid losing your session-specific security data, which is usually stored in `CLEAR_ON_DESELECT` RAM memory.

Non-MultiSelectable Applets

An error is returned to the terminal when an applet that is not designed to be aware of multiple channels is either selected more than once on different channels or is selected concurrently with other applets in the same package.

You can have several non-multiselectable applets operating simultaneously on different channels, as long as they do not interfere with each other's data while they are active. For example, you can open up to 4 channels and run a distinct applet on each as long as they do not interoperate. You can control their operation by multiplexing commands into the APDU communications channel. If the applets are independent of each other, then the results will be the same as if each of these applets were running one at a time, each in a separate session.

The MultiSelectable Interface

For an applet to be selectable on multiple channels at the same time, or to have another applet belonging to the same package selected simultaneously, it must implement the `javacard.framework.MultiSelectable` interface. Implementing this interface allows the applet to be informed when it has been selected more than once or when applets in the same package are already selected during applet activation.

Note:

If an applet in any package implements the `MultiSelectable` interface, then all applets in the package must also implement the `MultiSelectable` interface. It is not possible to have multiselectable and non-multiselectable applets in the same package.

The `MultiSelectable` interface contains a `select` and a `deselect` method to manage multiselectable applets. These methods are described in the following topics:

- Selection for MultiSelectable Applets
- Deselection for MultiSelectable Applets

Selection for MultiSelectable Applets

The `MultiSelectable` interface defines one method to be invoked instead of `Applet.select()` when the applet being selected, or any other applet in its package, is already selected on another logical channel:

```
public boolean MultiSelectable.select(boolean appInstAlreadySelected)
```

The `MultiSelectable.select(boolean)` method informs the applet instance if it is selected more than once on different channels, or if another applet in the same package is selected on another channel on any interface. The parameter `appInstAlreadySelected` is `true` if the applet is selected on a different channel. It is `false` if it is not selected. The method can return either `true` or `false` to accept or reject applet selection.

This method can be called as a result of issuing a `SELECT FILE` or a `MANAGE CHANNEL OPEN` APDU command used to select an applet. If the applet is not selected, then the `appInstAlreadySelected` parameter is passed as `false` to signal an applet activation event. If the applet is subsequently selected on another channel, `MultiSelectable.select(boolean)` is called again, but this time, the `appInstAlreadySelected` parameter is passed as `true`, to indicate that the applet is already active.

Deselection for MultiSelectable Applets

The `MultiSelectable` interface defines one method to be invoked instead of `Applet.deselect()` when the applet being deselected, or any other applet in its package, is already selected on another logical channel:

```
public void MultiSelectable.deselect(boolean appInstStillSelected)
```

The `MultiSelectable.deselect(boolean)` method informs the applet instance if it is being deselected on the logical channel while the same applet instance or another applet in the same package is still active on another channel on any interface. The parameter `appInstStillSelected` is `true` if the applet remains active on a different channel. It is `false` if it is not active on another channel, indicating that this is the last remaining active instance of the applet.

This method can be called as the result of a `MANAGE CHANNEL CLOSE` or a `SELECT FILE` APDU command. If the applet remains active on a different channel, the `appInstStillSelected` parameter is passed as `true`.

If the `MultiSelectable.deselect(boolean)` method is called, it means that either an instance of this applet or another applet from the same package remains active on another channel, so `CLEAR_ON_DESELECT` transients are not cleared.

Only when the last applet instance from the entire package is deselected does a call to `Applet.deselect()` occur, resulting in the erasure of `CLEAR_ON_DESELECT` transients.

Writing Applets for Concurrent Logical Channels

This section describes how to write a multiselectable applet that will perform various tasks based on whether it is selected. The code samples in this section show how to extend the applet to implement the `MultiSelectable` interface and how to implement the `MultiSelectable.select(boolean)` and `deselect(boolean)` methods. The code samples also show how to use the `Applet.select()` and `deselect()` methods to work with multiselectable applets.

To take advantage of multiple channel operation, an applet must implement the `javacard.framework.MultiSelectable` interface. For example:

```
public class SampleApplet extends Applet
    implements MultiSelectable {
    ...
    }
```

The new applet needs to provide implementation for the `MultiSelectable.select(boolean)` and `MultiSelectable.deselect(boolean)` methods. These methods are responsible for encoding the behavior that the applet needs during a selection event if either of the following situations occurs:

- The applet is already selected on a different channel
- One or more applets from the same package are also selected on different channels

The behavior to be encoded might include initializing applet state, accepting or rejecting the selection request, or clearing data structures in case of deselection:

```
public boolean select(boolean appInstAlreadySelected) {
    // Implement the logic to control applet selection
    // during a multiselection situation
    ...
}
public void deselect(boolean appInstStillSelected) {
    // Implement the logic to control applet deselection
    // during a multiselection situation
    ...
}
```

Note:

The applet is still required to implement the `Applet.select()` and the `Applet.deselect()` methods in addition to the `MultiSelectable` interface. These methods handle applet selection and deselection behavior when a multiselection situation does not happen.

The following topics describe how to perform tasks associated with writing applets for concurrent logical channels:

- MultiSelectable Applet Example
- Handling Channel Information on APDU Commands
- Writing ISO/IEC 7816-4:2013 Compliant Applets
- Non-MultiSelectable Applets and Shareable Objects
- ISO/IEC 7816-4:2013 Specific APDU Commands for Logical Channel Management

MultiSelectable Applet Example

In this example, the multiselectable applet, `SampleApplet`, must initialize the following two arrays of data when it is selected:

- An array of package data to be initialized when the first applet in the package becomes active
- An array of private applet data to be initialized upon applet instance activation

You can make these distinctions in your code because the `MultiSelectable` interface allows the applet to recognize the circumstances under which it is selected.

Also, the applet has the following requirements:

- Clear the package data once no applet in the package is active
- Clear the applet private data when the applet instance is deselected

The following methods are responsible for clearing and setting the data:

```
//dataType parameter as above
final static byte DATA_PRIVATE      = (byte)01;
final static byte DATA_PACKAGE      = (byte)02;
...
public void initData(byte[] dataArray, byte dataType) {
    ...
}
public void clearData(byte[] dataArray) {
    ...
}
```

To achieve the behavior specified above, you must modify the selection and deselection methods in your sample applet.

The code for `Applet.select()`, which is invoked when this applet is the first to become active in the package, can be implemented like this:

```
public boolean select() {

    // First applet to be selected in package, so
    // initialize package data and applet data
    initData(packageData, DATA_PACKAGE);
    initData(privateData, DATA_PRIVATE);
    return true;
}
```

Likewise, the implementation of the method `MultiSelectable.select(boolean)` must determine whether the applet is already active. According to its definition, this method is called when another applet within this package is active. `MultiSelectable.select(boolean)` can be implemented in a way that if `appInstAlreadySelected` is `false`, the applet private data can be initialized. For example:

```
public boolean select(boolean appInstAlreadySelected) {
    // If boolean parameter is false,
    // then we have applet activation
    // Otherwise, no applet activation occurs.
    if (appInstAlreadySelected == false) {
        // Initialize applet private data, upon activation
        initData(privateData, DATA_PRIVATE);
    }
    return true;
}
```

In the case of deselection, the applet data must be cleared. The method `MultiSelectable.deselect(boolean)` can be implemented in a way that it clears applet data only if the applet is no longer active. For example:

```
public void deselect(boolean appInstStillSelected) {

    // If boolean parameter is false, then applet is no longer
    // active.  It is O.K. to clear applet private data.
    if (appInstStillSelected == false) {
        clearData(privateData);
    }
}
```

If this applet is the last one to be deactivated from the package, it also must clear package data. This situation results in a call to `Applet.deselect()`. This method can be implemented like this:

```
public void deselect() {
    // This call means that the applet is no longer active and
    // that no other applet in the package is.  Data for both
    // applet and package must be cleared.
    clearData(packageData);
    clearData(privateData);
}
```

Handling Channel Information on APDU Commands

APDU commands follow the ISO/IEC 7816-4:2013 specifications to encode logical channel information in the CLA byte. The CLA byte encoding is divided into two spaces:

- Interindustry —Used by all ISO/IEC 7816-4:2013- defined commands
- Proprietary — Used by Java Card technology to encode application- specific commands

The CLA byte encoding is divided into two classes:

- Type 4 commands — Encode legacy ISO/IEC 7816-4 logical channel information
- Type 16 commands — Defined by the ISO/IEC 7816-4:2013 specification to encode information for additional 16 logical channels in the card.

Type 4 logical channels occupy the range of [`0`...`3`], while Type 16 logical channels go in the range of [`4`...`19`], that is, the value encoded in the CLA byte plus four, as it is used in `SELECT`, `MANAGE CHANNEL` and other proprietary or ISO commands.

However, a note of caution: while the `MANAGE CHANNEL` command CLA byte follows the encoding as described below, its P2 parameter does not. The logical channel numbers in its P2 parameter are correctly encoded in the range of [`0`...`19`].

The CLA byte encoding follows the following rules:

- Interindustry Space
- Proprietary Java Card Technology Space
- Logical Channels
- APDU Command Type Identification

Interindustry Space

CLA Remarks

`0x0X` Type 4, last or only command in chain

`0x1X` Type 4, not last command in chain (paired with `0x0X`)

`0x2X` Reserved for Future Use

`0x3X` Reserved for Future Use

`0x4X` Type 16, no SM, last or only command in chain

`0x5X` Type 16, no SM, not last command in chain (paired with `0x4X`)

`0x6X` Type 16, SM, last or only command in chain

`0x7X` Type 16, SM, not last command in chain (paired with `0x6X`)

The encoding details are as follows.

Type 4:

```
b8 b7 b6 b5 b4 b3 b2 b1
0  0  0  x  y  y  z  z
```

Type 16:

```
b8 b7 b6 b5 b4 b3 b2 b1
0  1  y  x  z  z  z  z
```

Notation:

`x` = Command Chaining bit

- `0` = last or only command
- `1` = command chaining

`y` = Secure Messaging indicator, see ISO7816-4:2003 section 6 for further information.

`z` = Logical channel indicator

Type 4 supports logical channels [0..3]

Type 16 supports logical channels [0..15], which are mapped to logical channels [4..19]

Proprietary Java Card Technology Space

CLA Remarks

`0x8X` Type 4, last or only command in chain

`0x9X` Type 4, not last command in chain (paired with 0x8X)

`0xAX` Type 4, last or only command in chain

`0xBX` Type 4, not last command in chain (paired with 0xAX)

`0xCX` Type 16, no SM, last or only command in chain

`0xDX` Type 16, no SM, not last command in chain (paired with 0xCX)

`0xEX` Type 16, SM, last or only command in chain

`0xFX` Type 16, SM, not last command in chain (paired with 0xEX)

The encoding details are as follows.

Type 4:

b8	b7	b6	b5	b4	b3	b2	b1
1	0	N/A	x	y	y	z	z

Type 16:

b8	b7	b6	b5	b4	b3	b2	b1
1	1	y	x	z	z	z	z

Logical Channels

When an APDU command is received, the card processes it and determines whether the command has logical channel information encoding. If logical channel information is encoded, then the card sends the APDU command to the respective channel. All other APDU commands are forwarded to the card's basic channel (`0`).

The `X` nibble is responsible for logical channels and secure message encoding. Only the two least significant bits of the nibble are used for channel encoding, which ranges from `0` to `3`. For example, the command `0x21` forwards the command to the card's basic channel (`0`), because the CLA byte with the nibble `0x2X` does not contain logical channel information.

Just as the deselection and selection mechanisms must be written to take into consideration a multiple-channel environment, it is important to write the `Applet.process()` method so that it handles channel information correctly. Due to the fact that some APDUs can be digitally signed, the APDU command is passed to the applet's `process` method as it is sent by the terminal. That means any logical channel information is not cleared and is passed intact to the applet. The applet must deal with this situation.

APDU Command Type Identification

To identify proprietary and interindustry commands, use the `isISOInterindustryCLA` method. This call returns `true` if the CLA byte encoding corresponds to the interindustry space, or `false` if it corresponds to the proprietary space.

```
...
//Applet's process method
public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
```

```
// check SELECT APDU command
if (apdu.isISOInterindustryCLA()) {
    if (Applet.selectingApplet()) {
        return;
    } else {
        ISOException.throwIt (ISO7816.SW_CLA_NOT_SUPPORTED);
    }
}
...
```

Writing ISO/IEC 7816-4:2013 Compliant Applets

If your applets must be compliant with the ISO/IEC 7816-4:2013 specification, then you must track the applet security state on each channel where it is active. Additionally, in the case of multiselectable applets, you must copy the state (including its security configuration) when you perform `MANAGE CHANNEL` commands from a channel other than the basic channel.

For example, applets might need to perform some sort of initialization upon activation, as well as cleanup procedures during deactivation. To do these tasks, a multiselectable applet might need to keep track of the channels on which it is being selected during a card session.

To track this information, you need to know the channel on which the task is being performed. Tracking is done by two methods in the Java Card API:

- `APDU` class: `public static byte getCLAChannel();`

  This method returns the origin channel where the command was issued. In case of `MANAGE CHANNEL` or `SELECT FILE` commands, if this method is called within the `Applet.select()`, `MultiSelectable.select(boolean)`, `Applet.deselect()`, or `MultiSelectable.deselect(boolean)` methods, it returns the APDU command logical channel specified in the CLA byte.

- `JCSystem` class: `public static byte getAssignedChannel();`

  This method returns the channel of the currently selected applet. In case of a `MANAGE CHANNEL` command, if this method is invoked inside the `Applet.select()`, `MultiSelectable.select(boolean)`, `Applet.deselect()`, or `MultiSelectable.deselect(boolean)` methods, it returns the channel where the applet to be selected or deselected is assigned to run.

ISO/IEC 7816-4:2013 Compliant Applet Example

This example demonstrates how to copy the security state from the applet selected in the origin channel into the new channel.

In this example, the state information is stored in the array `appState` inside the applet:

```
StateObj appState[MAX_CHANNELS];    // Holds the security state
                                        // for each logical channel
```

You can use the `APDU.getCLAChannel()` and the `JCSystem.getAssignedChannel()` methods to identify if the applet selection case corresponds to an ISO/IEC 7816-4 case where the security state needs to be copied.

Note:

If such an event occurs, it will also be a multiselection situation, where the applet is also selected on the newly opened channel.

In this example, the code to identify the applet selection case is included in the implementation of the `MultiSelectable.select(boolean)` method:

```
// Applet's select method
public boolean select(boolean appInstAlreadySelected) {
        ...
        // Obtain logical channels information
        // This call returns the channel where
        // the command was issued
        byte origChannel = APDU.getCLAChannel();
        // This call returns the channel where the applet is being
        // selected
        byte targetChannel = JCSystem.getAssignedChannel();
        if (origChannel == targetChannel) {
            // This is a SELECT FILE command.
            // Do processing here.
            ...
        }
        if (origChannel == 0) {
            // This is a MANAGE CHANNEL command from channel 0.
            // ISO 7816-4 state sharing case does not
            // apply here.
            // Do processing here.
            ...
        } else {
            // Since origChannel != 0, the special
            // ISO 7816-4 case applies.
            // Copy security state from origin channel
            // to assigned logical channel.
            appState[targetChannel] = appState[origChannel];
            // Do further processing here
            ...
        }
        ...
    }
```

Refer to the API documentation located at https://docs.oracle.com/en/java/javacard/3.2/jcapi/api_classic/index.html for more information about the APIs.

Non-MultiSelectable Applets and Shareable Objects

Applets that implement `MultiSelectable` are designed to handle calls to Shareable objects across packages when several applets are active on different logical channels. In contrast, an applet that does not implement `MultiSelectable` assumes that it is uniquely selected and its owned objects will not be modified via Shareable interface objects while it is selected. Only when the non-multiselectable applet is in a deselected state can other applets modify its internal data structures.

When you interact with applets that do not implement `MultiSelectable`:

- It is not possible to invoke methods of a Shareable object belonging to a non-multiselectable applet when an applet, belonging to the same group context, is active
- It is not possible to invoke methods of a Shareable object belonging to a non-multiselectable applet when an applet, belonging to the same group context, is active

ISO/IEC 7816-4:2013 Specific APDU Commands for Logical Channel Management

There are two ISO-specific APDU commands that you can use to work with logical channels in a smart card:

- `SELECT FILE` — This command selects the specified applet on the specified channel number. The channel number can be from `0` to `3` and is specified in the lower two bits of the CLA byte. If the channel is closed, it is opened and the specified applet is selected on the channel. `SELECT FILE` commands are forwarded to the newly selected applet
- `MANAGE CHANNEL` — This command can be used to open a new channel from another channel or close it. It allows you to specify the channel to be used or to allow the smart card to select the channel. Like `SELECT FILE`, this command uses the lower two bits of the CLA byte to specify the channel number. `MANAGE CHANNEL` commands are not forwarded to the applet

When you work with these commands, keep the following guidelines in mind:

- Origin logical channel values are encoded in the two least significant bits of the CLA byte
- Logical channel values have a valid range of [`0`..`19`] only
- Logical channel `0` is known as the *basic channel*, and it cannot be closed
- At card reset, the basic channel (channel `0`) is open. All other channels (`1`, `2`, ...`19`) are closed

The `MANAGE CHANNEL` and `SELECT FILE` commands are read by the Java Card RE dispatcher, which performs the functions specified by the commands, including the following:

- Managing logical channels
- Deselecting applets
- Selecting applets

MANAGE CHANNEL OPEN

In response to the `MANAGE CHANNEL OPEN` command, the dispatcher follows this procedure:

1. If the origin channel is not open, an error is returned.
2. Determines whether the channel is open or closed. If the channel is open, an error is returned.
3. Opens the channel.
4. If the origin channel is `0`, the default applet (if there is one) is selected in the new channel.
5. If the origin channel is not `0`, the selected applet on the origin channel becomes the selected applet in new channel.

This MANAGE CHANNEL OPEN command opens a new channel from channel encoded in Q:

CLA	INS	P1	P2	Lc	Data	Le	Data	SW1	SW2
0xQ	0x70	00	00	0	-	1	0x0R	0x90	00

:

CLA	INS	P1	P2	Lc	Data	Le	SW1	SW2
0xQ	0x70	00	0xR	0	-	0	0x90	00

This command produces the following results:

- If channel encoded in Q is the basic channel (channel 0), the card's default applet is selected on channel encoded in R. No applet is selected if no default applet is defined
- If channel encoded in Q is other than the basic channel (channels 1, 2, ...19), the selected applet on channel encoded in Q becomes the current applet selected on channel R
- The applet on channel encoded in R can either accept or reject selection

This command returns an error under the following circumstances:

- The applet does not implement the javacard.framework.MultiSelectable interface, when an attempt to select the applet in more than one channel takes place
- The applet rejects selection or throws exception
- No channel is available
- Channel encoded in Q is not open

MANAGE CHANNEL CLOSE

In response to the MANAGE CHANNEL CLOSE command, the dispatcher follows this procedure:

1. If the origin channel is not open, an error is returned.
2. If the channel to be closed is 0, an error is returned.
3. If the channel to be closed is not open or not available, a warning is thrown.
4. Deselects the applet in the channel to be closed.
5. Closes the logical channel.

This MANAGE CHANNEL CLOSE command closes channel R from channel Q:

CLA	INS	P1	P2	Lc	Data	Le	SW1	SW2
0xQ	0x70	0x80	0xR	0	-	0	0x90	00

This command closes channel R. Channel R must not be the basic channel (it can be channel 1, 2, ...19 only).

This command returns an error in the following circumstances:

- Channel encoded in `R` is the basic channel
- Channel encoded in `Q` is not open

It returns a warning if channel R is not open

SELECT FILE

In response to the `SELECT FILE` command, the dispatcher follows this procedure:

1. If the specified channel is closed, open the channel.
2. Deselect currently selected applet in channel if needed.
3. Select specified applet in the channel.

This `SELECT FILE` command selects an applet on channel `R`:

CLA	INS	P1	P2	Lc	Data	Le	SW1	SW2
`0x0R`	`0xA4`	`0x04`	`0x00`	(AID len)	(AID)	`0`	`0x90`	`00`

This command produces the following results:

- Channel encoded in `R` can be any channel (opened or unopened), including the basic channel
- The applet identified in the Data section becomes the selected applet on channel `R`
- If channel encoded in `R` is not open, this command opens channel `R`
- If channel encoded in `R` is open, this command changes the selected applet in the channel to the one specified

This command returns an error in the following circumstances:

- The applet cannot be found or is not available. The current applet is left selected and an error is returned
- An active applet belonging to the same package does not implement the `javacard.framework.MultiSelectable` interface, or if the applet to be selected does not implement this interface
- Channel encoded in `R` is not available

8
Using Extended APDU

This chapter describes the Extended APDU and how it can be used to allow large amounts of data to be sent to the card, processed appropriately, and sent back to the terminal. The Extended APDU feature is especially beneficial to applications that deal with large amounts of information, such as signature verification, biometrics verification, and image storage and retrieval. These are more easily implemented if the underlying transport protocol is T=1. Applets developed for T=0 cards need special logic and care to work correctly.

This chapter includes the following topics:

- Extended APDU Nominal Cases
- Extended APDU Format
- Extended APDU Limits
- Creating an Applet That Can Send and Receive Extended Length APDUs

Extended APDU Nominal Cases

The ISO/IEC 7816-4:2013 specification defines an extended APDU as any APDU whose payload data, response data or expected data length exceeds the 256 byte limit. Therefore, the four traditional cases are redefined as follows:

- Case 1. As in short length, this case is not affected.
- Case 2S. The legacy Case 2 from previous Java Card technology releases. LE has a value of 1 to 255.
- Case 2E. The extended version of Case 2S, where LE is greater than 255.
- Case 3S. The legacy Case 3. LC is less than 256 bytes of data, and LE is zero.
- Case 3E. The extended version of Case 3, where LC is greater than 255, and LE is zero.
- Case 4S. The legacy Case 4. LC and LE are less than 256 bytes of data.
- Case 4E. The extended version of Case 4. LC or LE are greater than 256 bytes of data.

Extended APDU Format

Any APDU classified as extended must follow the format defined by ISO/IEC 7816-4:2013 for extended length APDU and summerized in the following table.

Table 8-1 Extended APDU Format

Field	Description	Number of Bytes
Command Header	Class byte CLA	1
Command Header	Instruction byte INS	1
Command Header	Parameter bytes P1- P2	2

Table 8-1 (Cont.) Extended APDU Format

Field	Description	Number of Bytes
LC Field	Absent for Nc = 0. Present for Nc > 0	0, 1, or 3
Data Field	Absent if Nc = 0, present if Nc >0	Nc
LE Field	Absent for Ne = 0, present for Ne > 0	0, 1, 2 or 3
Response Data	Absent if Nr = 0, present if Nr >0	Nr (max. Ne)
Response Status	Status bytes SW1 SW2	2

Notation

Nc = command data length

Ne = expected response data length

Nr = actual response data length

The encoding rules are defined as:

For LC:

- If LC field is absent, Nc = 0.
- If LC is present as one byte with values between `01` and `FF`, then Nc = 1..255 accordingly, and it will be a short field.
- If LC is present as an extended field, then it will be three bytes in length: byte one will be `00`, bytes two and three will contain a 16-bit value representing the length of the data Nc with values between 1 and 65535.

For LE:

- If LE is absent, Ne = 0.
- If LE is one byte:
  - A value between `01` and `FF` will indicate Ne = 1..255.
  - A value of `00` will indicate Ne = 256.

If LE is an extended field:

- LC and LE must be in the same format.
- An LE field value between `0001` and `FFFF` will indicate Ne = 1..65535.
- An LE field value of `0000` will indicate Ne= 65536.

Extended APDU Limits

The Java Card, Version 3.2 platform supports extended APDUs with some limitations. Because the platform defines all of its mandatory API in terms of short data length, the values of LC and LE are limited to short positive values. That is, LC and LE have a range of `0`..`32`,`767`. Lengths of `32`,`768` and beyond are not supported by the Java Card, Version 3.2 platform at this time.

Creating an Applet That Can Send and Receive Extended Length APDUs

To create an applet that can send and receive extended length APDUs:

1. Implement the `javacardx.apdu.ExtendedLength` interface in your applet:

```
...
import javacard.framework.*;
import javacardx.apdu.ExtendedLength;
...
public MyApplet extends Applet implements
ExtendedLength {
...
}
```

2. Write your applet and `Applet.process(..)` method as you would with any other applets. For consistency, it is advisable that your `process(..)` code begin like the one below:

```
public void process(APDU apdu) {
   byte[] buffer = apdu.getBuffer();

   if (apdu.isISOInterindustryCLA()) {
       if (this.selectingApplet()) {
           return;
       } else {
           ISOException.throwIt (ISO7816.SW_CLA_NOT_SUPPORTED);
       }
   }

   switch (buffer[ISO7816.OFFSET_INS]) {
   case CHOICE_1:
       ...
       return;
   case CHOICE_2:
       ...
       ...
   default:
       ISOException.throwIt (ISO7816.SW_INS_NOT_SUPPORTED);
   }
}
```

3. For cases 3S, 4S, 3E and 4E, write the method to handle incoming data. Use the API so that your applet properly handles extended, as well as non-extended, cases.

```
void receiveData(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    short LC = apdu.getIncomingLength();

    short recvLen = apdu.setIncomingAndReceive();
```

```
        short dataOffset = apdu.getOffsetCdata();

        while (recvLen > 0) {
            ...
            [process data in buffer[dataOffset]...]
                    ...
                    recvLen = apdu.receiveBytes(dataOffset);
        }
        // Done
    }
```

4. For case 2S, 2E, write the method handling data output. A method could look something like this:

```
void sendData(APDU apdu) {
   byte[] buffer = apdu.getBuffer();

   short LE = apdu.setOutgoing();
   short toSend = ...

   if (LE != toSend) {
       apdu.setOutgoingLength(toSend);
   }

   while (toSend > 0) {
       ...
       [prepare data to send in APDU buffer]
               ...
               apdu.sendBytes(dataOffset, sentLen);
               toSend -= sentLen;
  }
  // Done
}
```

9 Java Card Development Kit Accessibility Information

This topic details the Java Card Development Kit features that support accessibility.

Topics:

- Access to Java Card Development Kit Support
- Java Card Development Kit Features that Support Accessibility
- Keyboard Navigation
- Documentation Accessibility Features

Access to Java Card Development Kit Support

The Java Card Development Kit customers have access to electronic support through email with their assigned support engineer provided by the Java Card Licensee Engineering (JLE) organization.

Hearing impaired customers in the U.S. who wish to speak to their assigned support engineer can use the telecommunications relay service (TRS). Information about the TRS is available at https://www.fcc.gov/consumers/guides/telecommunications-relay-service-trs. International hearing impaired customers must use the TRS at +1.605.224.1837.

Java Card Development Kit Features that Support Accessibility

Oracle's goal is to ensure that disabled users of our products can perform the same tasks, and access the same functionality as other users.

Java Card Development Kit supports the following accessibility features:

- Can be operated using only the keyboard
- Communicates all information independent of color
- Time Based Media is not used
- Images of text are not used
- Moving, blinking, or scrolling content is not used
- Doesn’t disrupt platform accessibility features such as Sticky Keys, High Contrast, and Large Fonts
- Provides online documentation in an accessible format

Keyboard Navigation

Java Card Development Kit uses standard navigation keys.

Documentation Accessibility Features

Java Card Development Kit documentation supports the following accessibility features:

- The documents are available in the HTML format to give maximum opportunity for the users to apply screen-reader software technology.
- The images in the documents are provided with alternative text so that users with vision impairments can understand the contents of the images.

Part III

Java Card Eclipse Plug-in

This part of the user guide describes how to use the Java Card Eclipse Plug-in to create a Java Card project, Java Card applet, and to debug an applet.

Accessibility information for Java Card Eclipse Plug-in

Java Card Eclipse Plug-in provides a wide range of features that support accessibility. Oracle is committed to creating products, services, and supporting documentation that is accessible to the disabled community. The Java Card Eclipse Plug-in is executed through Eclipse and inherits all the accessibility features provided by Eclipse. For more information please visit https://wiki.eclipse.org/Accessibility.

This chapter contains the following section:

- Using the Java Card Eclipse Plug-in

10
Using the Java Card Eclipse Plug-in

This chapter contains the following topics:

Topics:

- Creating a Java Card Project Using the New Java Card Project Wizard
- Creating a Java Card Applet Using the Default Source Template
- Creating a CAP File in a Java Card Project
- Adding a Java Card Package to a CAP File
- Adding a Java Card Applet to a Java Card Package
- Adding a Java Card Static Resource to a CAP File
- Managing Applets and Sending APDU Commands
- Debugging a Java Card Applet in Eclipse Plug-in

Creating a Java Card Project Using the New Java Card Project Wizard

To create a new Java Card Project, based on a default template, use the New Java Card Project wizard:

1. Click the **File** menu, and then select **New** and **Other…**.
2. In the Other… dialog, under **Oracle Java Card SDK**, select **Java Card Project**.
3. In the first page of the wizard, configure the following Java Card specific sections:
   - **Runtime Environment** - Select the Java Card Platform that you want to use in the project. The platform and devices are selected from the existing configuration. If you have not configured the platform and devices, you can use a link that opens the platform and devices settings page.
   - **Java Card Tools** - Configure the Java Card tools bundle path, if not configured already.
   - **Application** - Configure a Java Card package and/or applet with names and AIDs. When an applet is configured, a default Applet source template is created.
4. Click **Finish**.

   A new Java Card project is created in the workspace containing the default CAP file configuration. However, this is valid only if the application is configured in Step 3.

Changing the Runtime Environment for the Java Card Project

To change the runtime environment:

1. Right-click on the Java Card project and select **Java Card** and **Runtime Setting**.
2. Select a platform from the **Platform** section. If a platform is not configured, click the link that opens the Java Card platforms and device global settings pages.
3. In the **Device** section, select a configured device for the selected platform. If a device is not configured, click the link that opens the Java Card platforms and device global settings page.

Creating a Java Card Applet Using the Default Source Template

To create Java Card Applets use the default source template of the Eclipse Plug-in:

1. Select a Java Card Project, click the **File** menu, and select **New** and **Other…**.
2. In the Other… dialog, expand **Oracle Java Card SDK** and select **Java Card Applet**.
3. In the **Package** section, click **Browse** and select the Java package where you want to add the Applet source template.
4. In the **Applet name** field, enter a Java class compliant name.
5. Click **Finish**.

   The Java Card applet is added to the package.

Creating a CAP File in a Java Card Project

To create deliverables in a Java Card project, you must create and configure CAP files.

To create a CAP file:

1. Select a Java Card Project, click the **File** menu, and select **New** and **Other…**.
2. In the Other… dialog, expand **Oracle Java Card SDK** and select the **Java Card CAP File**.
3. In the Select CAP file type page, select either **Compact CAP file** or **Extended CAP file**. Each type of the CAP file has a specific function in the Java Card specification. It is important to note the following:
   - A compact CAP file can have only one Java Card package configured.
   - An extended CAP file can be used only with the 3.1.0 or greater Java Card platform versions.
4. In the Select CAP file settings page, configure the `converter` tool options for the build:
   a. In the **CAP File AID** section, enter a unique CAP file name and CAP file AID. If the CAP file is a compact CAP file, then the AID field is disabled. This is because, the CAP file AID is inherited from the Java Card package that is configured.
   b. In the **Converter** section, configure the `converter` tool:
      - In the **Options** section, enter values for the CAP file version, target platform, and all the flags. If a CAP file is a compact CAP file, then the

version is inherited from the Java Card package that is configured. If a CAP file is an extended CAP file, then the target platform must be 3.1.0 or greater and the mask flag option is disabled.

- In the **Export Path** section, add the directories in which the `converter` tool searches for the export files. The `verifier` tool uses the paths added in this step during the build process. If the paths are added from the project, relative paths are generated.
- In the **CAP Signing** section, configure the CAP sign feature of the `converter` tool.

Note:

It is **important** to note that passwords are stored in plain text in the metadata of your workspace. They are readable by anyone with direct access to this computer.

5. Click **Finish**.

   A dialog appears prompting you to confirm if the Java Card package needs to be configured for the CAP file you just created. If you click **Yes**, the Package Configuration dialog appears. If you click **No**, the wizard closes and a new configuration file to be included in the build is created.

Managing CAP File Configurations

To manage the CAP file configurations list and to edit a CAP file's configuration, use the CAP file project settings:

1. Right-click on the Java Card project and select the **Java Card** and **CAP Files Settings**.
2. In the Java Card CAP Files page, to manage the CAP files used in the build, use the following options:
   a. Click **Add** to create a new CAP file using the wizard.
   b. Click **Edit** to edit a CAP file's configuration using the wizard with all fields set to the previous values.
   c. Click **Delete** to delete a CAP file's configuration from the build.

Adding a Java Card Package to a CAP File

To include a CAP file in the build, it must be in the configured state. A Cconfigured CAP file is a CAP file that has at least one Java Card package added to it.

To add a Java Card package to a CAP file:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
2. In the Java Card CAP Files page, select a CAP file from the list.
3. Click **Add new package**.

> **Note:**
>
> If the CAP file is a compact CAP file and it is already configured, then **Add new package** is disabled. A compact CAP file can contain a maximum of one package only.

4. In the Configure Java Card package dialog, configure the following `converter` tool specific parameters:
   a. Next to the **Package Name** field, click **Browse…** and select a Java package from the project
   b. Enter values for the Java Card package AID and version fields.
   c. Verify the `converter` tool flags. The **Private** flag is available only if the CAP files are extended CAP files.
5. Click **OK**.

   A dialog appears prompting you to confirm if a Java Card Applet needs to be configured for the package that you just created. If you click **Yes**, the Java Card Applet configuration dialog appears. If you click **No**, the dialog closes, the CAP file configuration is updated, and the project is rebuilt.

Managing the Java Card Package

To manage the Java Card package added to a CAP file and to edit the Java Card package, use the CAP file project settings:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Setting**.
2. In the CAP Files Settings page, click the arrow to the left of the **Java Card CAP Files**.
3. Select **Java Card Packages**.

   A list with package names appears.
4. Select a CAP file from the combo list.

   The list is populated with the Java Card packages configured for the selected CAP file.
5. To manage the Java Card packages configured for the selected CAP file, perform the following tasks:
   a. Click **Add** to add a new Java Card package to the selected CAP file. If the CAP file is a configured compact file, this button is disabled.
   b. Click **Edit** to edit an already configured Java Card package.
   c. Click **Delete** to delete a Java Card package from the selected CAP file.

Adding a Java Card Applet to a Java Card Package

If a CAP file is an applet CAP file, then you need to configure the applets that are contained in it. A `Java Card Applet` is a class contained in a Java Card package.

To add a Java Card Applet to a Java Card package:

1. Right-click on the Java Card project and select the **Java Card** and **CAP Files Settings**.
2. In the CAP Files Settings page, click the arrow to the left of the **Java Card CAP Files**.
3. Select **Java Card Packages** and the CAP file in which you want to add the Applet.
4. Select the package that contains the Applet you want to add.
5. Click **Add new applet**.
6. In the Configure Java Card Applet window, set the `converter` tool parameters for the applets:
   a. In the **Applet** section, click **Browse…** and select the applet class from the list.
   b. In the **Applet AID** section, enter a value for the PIX part of the AID of the Java Card Applet. The RID part of the AID is automatically populated based on the Java Card package configuration.
7. Click **OK**.

   The window closes, the CAP file configuration is updated, and the project is rebuilt.

Managing Java Card Applets

To manage Java Card Applets and to edit them, use the CAP file project settings:

1. Right-click on the Java Card project and select the **Java Card** and **CAP Files Settings**.
2. In the **CAP Files Settings** page, click the arrow to the left of the **Java Card CAP Files**.
3. Click the arrow to the left of the **Java Card Packages**.
4. Select **Java Card Applets**.

   A list with Applets appears.
5. Select a CAP file and Java Card package combination from the combo list.

   The list is populated with the Java Card Applets configured for the selected combination.
6. To manage the Java Card Applets configured for the selected combination, perform the following tasks:
   a. Click **Add** to add a new Java Card Applet to the selected CAP file or package combination.
   b. Click **Edit** to edit an already configured Java Card Applet.
   c. Click **Delete** to delete a Java Card Applet from the selected CAP file or package combination.

Adding a Java Card Static Resource to a CAP File

With Java Card, Version 3.1.0 and later, you can add static resources to a CAP file while loading.

To add a Java Card static resource to a CAP file:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Settings**.
2. In the Java Card CAP Files page, select a CAP file from the list.
3. Click **Add new static resource**.

4. In the **Configure Java Card static resource** dialog, configure the `converter` tool static resource-specific parameters:
   a. In the **Static Resource ID field**, enter a unique integer number.
   b. In the **Static Resource file path** section, click **Browse…** and select the file that you want to add as a static resource to a CAP file. If the path is inside the project, a relative path is generated.
5. Click **OK**.

   The **Configure Java Card static resource** dialog closes, the CAP file configuration is updated, and the project is rebuilt.

Managing Java Card Static Resources

To manage Java Card static resources list added to a CAP file, and to edit the Java Card static resource, use the CAP file project settings:

1. Right-click on the Java Card project and select **Java Card** and **CAP Files Setting**.
2. In the CAP Files Settings page, click the arrow to the left of the **Java Card CAP Files**.
3. Select **Java Card Static Resources**.

   A list with static resources appears.
4. Select a CAP file from the combo list.

   The list is populated with Java Card static resources configured for the selected CAP file.
5. To manage the Java Card static resources configured for the selected CAP file, perform the following tasks:
   a. Click **Add** to add a new Java Card static resource to the selected CAP file.
   b. Click **Edit** to edit an already configured Java Card static resource.
   c. Click **Delete** to delete a Java Card static resource from the selected CAP file.

Managing Applets and Sending APDU Commands

To manage the Java Card applets and send APDU commands use Java Card View and Device Console.

1. Start the Simulator by right-clicking on `Sample_Device` in **Java Card View** and selecting **Start**.
2. The console opens with the output from simulator, 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.
3. One simple way to test if the console is running is to type the help command at the prompt:

   help;
4. You should see the available commands.
5. To load and install a built package, click on the drop down menu from `Sample_Device` console. The available commands for all imported projects will be displayed. If connect wasn't done automatically, by default it's done automatically,

click on **Connect**. In the list are displayed all projects from current workspace with device set for `Sample_Device`. Under each project submenu, you can find all CAP files assigned to this project and for each CAP file are listed all available commands for specific CAP file. Click first on **Load CAP_AID/Package_AID** and wait for command execution, then click **on Install CAP_AID/Package_AID Applet_AID (package.AppletName)**.

All available commands:

- **Connect** – Performs a card connection to the simulator.
- **Disconnect** - Performs a card disconnection from the Simulator. If the Simulator is launched from the Plug-in, you are able to reconnect pressing the connect or writing in console `Connect;` again.
- **List applets** – Displays AID for installed applets on card.

For each CAP file from each Java Card Project submenu, you will have following commands:

- **Load CAP_AID/Package_AID** – Loads the applet on the card.
- **Unload CAP_AID/Package_AID** – Unloads the `Applet_Name` from card.
- **Load, Install, Select CAP_AID/Package_AID Applet_AID** – Loads, installs and selects the `Applet_Name` from card in one command.
- **Custom Install CAP_AID/Package_AID Applet_AID (package.AppletName)** – Install the loaded applet on the card with Application Specific Parameters and/or with different Applet Instance AID (After this command is executed with different Applet Instance AID, a new `Select` and `Uninstall` command will be generated for new Applet Instance AID).
- **Install CAP_AID/Package_AID Applet_AID (package.AppletName)** – Install the loaded applet on the card.
- **Select CAP_AID/Package_AID Applet_AID (package.AppletName)** – Select the `Applet_Name`.
- **Uninstall CAP_AID/Package_AID Applet_AID (package.AppletName)** – Uninstalls the `Applet_Name` from card.

Run Configuration

Run Configuration can be used to automate how commands are run. You can specify whether simulator shall be connected or re-connected and provide a list of commands to be executed.

The Run Configuration has option to start simulator, inside eclipse, for you. This option can be disabled and just execute selected commands, the plugin will connect to an already running simulator. The running simulator must match the configuration done in selected device (ex. `Sample_Device`) for current project (**Communication type**, **Port number** and **Config file**).

Managing HelloWorld sample

These steps show you how to manage 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.

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

- Right-click on the `Sample_Device` and press Start
- If **Connect** command is not sent automatically, from the drop-down button within the console press **Connect**

Note:

If the `Sample_Device` console is not selected automatically. From **Console View**, locate the `Sample_Device` console under the **Display Selected Console** drop-down, in the **Console View** toolbar.

- Check output of the console for successful connect by checking the response APDU (SW:9000)
- After a successful connect the drop-down menu will fill with the commands needed for managing the HelloWorld sample
- From the drop-down menu select **HelloWorld > CAP: HelloWorld > Load, Install, Select A00000006203010C01 A00000006203010C0101 (com.oracle.jcclassic.samples.helloworld.HelloWorld)**
- Check that all commands were executed successfully. (SW:9000)
- Send the following APDU, from the Sample_Device Console: `CMD>0x80 0x20 0x00 0x00 0x05 0x31 0x32 0x33 0x34 0x35;`
- The response should look like this: `[APDU-R] 80200000053132333435 SW:9000`

Debugging a Java Card Applet in Eclipse Plug-in

From Eclipse, you can run the debug proxy to set breakpoints, get or set variable values, and debug a library.

These steps are an overview of how to debug an application from Eclipse.

The Java Card Plug-in for Eclipse must already be installed.

1. Create (or import) your Java Card project. Make sure that debugging information is generated when the project is built.

Note:

The debugging information is generated only if you select the **Enable generation of debugging information** check box in the CAP file settings.

2. Create a new Java Card Project Debug configuration and select the **Java Card** tab. Perform the following tasks:
   a. Specify commands to be executed when the simulator starts. The commands run in an order represented in the table. The connect command is mandatory in order to debug the applet and plug-in performs it automatically.
   b. Add cap files for the applet and imported libraries, which include the code that you want to debug. These not only include CAP files generated in the `deliverables` directory, but also the CAP files that you generated.

> **Note:**
>
> The CAP files generated by current project, will be added automatically.

3. Once the debug session starts, `simulator` starts in debug mode, the commands(s) are executed, the debug proxy is started, and the Eclipse debugger connects to the debug proxy.

   You can experiment with the debug perspective and look at the debug console for debug proxy output.

4. Set breakpoints and execute scripts.

Debug Configuration

Debug Configuration can be used to automate how commands are run before sending the commands to hit the breakpoints. You can specify a list of commands to be executed.

In the JC Debug Configuration, under **Connect** tab -> **Connection Properties** -> **Port** the value must match the **Proxy to IDE port** from the device properties which, by default, both are 8000.

The Debug Configuration has an option to start the simulator, inside Eclipse, for you. This option can be disabled to just debug the applet and execute selected commands, the plug-in will connect to an already running simulator. The running simulator must match the configuration made in the selected device (ex. `Sample_Device`) for the current project (**Communication type**, **Port number**, **Config file**, **Debug Port**).

Debugging HelloWorld Sample from Eclipse

These steps show you how to debug the `HelloWorld` sample. The Java Card Plug-in for Eclipse must already be installed.

Start Eclipse. `Sample_Platform` and `Sample_Device` must already be created.

1. Click the **File** menu, select **Import >General > Projects from Folder or Archive**, and select the `applet` directory from the HelloWorld project to import the HelloWorld Java Card project into your workspace. If the build doesn't start automatically, start it manually.
2. Make sure debugging information generation is enabled for the `HelloWorld` package:
   a. Right-click on the imported project and select **Java Card** and **CAP Files Settings**.
   b. Select the `HelloWorld` CAP file and click **Edit**.
   c. In the new wizard, click **Next** on the first page.
   d. In the second page, select the **Enable generation of debugging information** check box.
   e. Click **Finish** and **Apply**, and close the wizard.
3. Create a new debug configuration:
   a. Right-click on the `HelloWorld` project in the Package Explorer and select **Debug As** and **Debug Configurations**.
   b. In the Debug Configurations dialog, double-click **Java Card Project Debug** (in the list). This will create a new debug configuration named HelloWorld.

c. Select the **Java Card** tab.

d. Select the **Start simulator in debug mode…** and **Start debug proxy…** check boxes.

e. Click **Add command**. Browse to the `HelloWorld` project directory and select the `Load command` file. This script will install the applet without creating an applet instance.

f. Click **Debug**.

The debug configuration starts. First, `simulator` is started in debug mode, then the commands are executed, the debug proxy is started and, finally, the Eclipse debugger connects to the debug proxy.

4. The Confirm Perspective Switch dialog appears, asking if you want to open the Debug perspective. You may choose to open it, depending on your preference.

   The Debug console shows output from the debug proxy.

5. In the Package Explorer, locate `HelloWorld.java` and open it. Set two breakpoints: one in the `install()` method of the applet, the other in the beginning of the `process()` method.

   There are several ways to set a breakpoint in Eclipse. In the source code editor, position the cursor on the desired line and do one of the following:

   a. Double-click the left most space on the source code line (the line number will be to the right).

   b. Press **Ctrl + Shift + B** to toggle the breakpoint (the type of breakpoint will be selected automatically depending on the source code).

   c. Select a specific breakpoint to toggle from the **Run** menu.

6. Execute the two remaining scripts in the order that they appear in the Package Explorer in the **Select Script** dropdown from **Sample_Device Console** (Install and Select command).

   After each script is run, execution will suspend on the corresponding breakpoint. `Step*` and `resume` debugger commands can be used to resume applet code execution.

Part IV
Appendices

The following appendices contain a Java Card assembly syntax example and a description of additional, optional Ant tasks:

- Annex - Using Cryptography Extensions
- Annex - Application Management

A

Annex - Using Cryptography Extensions

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

This chapter contains the following sections:

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

Overview of Using Cryptography Extensions

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

- Generate message digests using `MD5`, `RIPEMD160`, `SHA1` and `SHA2` algorithms
- Generate cryptographic keys on Java Card technology-compliant smart cards for use in the `ECC`, `DSA` and `RSA` algorithms
- Set cryptographic keys on Java Card technology-compliant smart cards for use in the `AES`, `DES`, `3DES`, `HMAC`, `ECC`, `DSA`, and `RSA` algorithms
- Encrypt and decrypt data with the keys using the `AES`, `DES`, `3DES`, and `RSA` algorithms
- Encrypt and decrypt data and associated data with `AES` keys using `AEAD` algorithms
- Generate and verify signatures using `MAC`, `CMAC`, `HMAC`,`DSA`, `ECDSA`, and `RSA` algorithms.
- Generate and verify signatures with message recovery using `RSA` algorithm
- Generate sequences of random bytes
- Perform key-agreement with `ECC` algorithm
- Generate checksums with `CRC` algorithms
- Support padding schemes for signatures and cipher operations
- Generate derived data using `KDF` in Counter mode and `HKDF` for `TLSv1.2`, `TLSv1.3` and `DTLSv1.3` algorithms

Note:

`DES` is also known as single-key `DES`. `3DES` is also known as triple-`DES`.

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

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

- For `RSA` — "*RSASSA-PSS (Probabilistic Signature Scheme padding. Signature Scheme)*": PKCS#1-PSS scheme (IEEE 1363-2000), PKCS#1-OAEP scheme (IEEE 1363-2000)
- For `RSA` — "*RSA-OAEP (Optimal Asymmetric Encryption Padding) Encryption Scheme*"
- For `RSA` - *Signature with message recovery*: ISO/IEC 9796-2
- For `AES` — "*Advanced Encryption Standard (AES)*" FIPs Publication 197: https://www.nist.gov/itl
- For ECDSA —*"Digital Signature Standard (DSS)" FIPS PUB 186-2:* https://csrc.nist.gov
- For ECB, CBC, CFB — *"Recommendation for Block Cipher Modes of Operations " NIST SP 800-38A:* https://csrc.nist.gov/pubs/sp/800/38/a/final
- For AES-XTS — *"IEEE Standard for Cryptographic Protection of Data on Block-Oriented " Storage Device" IEEE Std 1619-2018* https://standards.ieee.org
- For ISO-9797 padding methods — "Information technology – Security techniques – Message Authentication Codes (MACs) Part 1: Mechanics, using a block cipher" ISO(IEC-9797-1:2011: https://www.iso.org
- For PKCS#5 padding — "PKCS#5: Password-Based Cryptography Specification Version 2.0" https://datatracker.ietf.org/doc/html/rfc2898
- For Checksum — *"Informationtechnology—Telecommunications and information exchange between systems—High-level data link control (HDLC) procedures"* ISO/IEC-13239:2002 (replaces ISO-3309): https://www.iso.org
- For `SHA224`, `SHA256`, `SHA384` and `SHA521` — *"Secure Hash Standard"*, FIPS Publication 180-2: https://www.nist.gov/itl
- For RIPEMD-160 — *"Information technology – Hash functions – Part 3: Dedicated hash functions"* ISO/IEC 10118-3:2018: https://www.iso.org
- For HMAC with SHA1 or SHA256 — *"Keyed-HashingforMessageAuthentication"*, RFC-2104
- For `HKDF` — *Expand-Label* of `TLSv1.3`: IETF RFC 8446 and `DTLS1.3`: IETF RFC 9147
- For `DSA` — *"Digital Signature Algorithm"*, Standard, NIST FIPS 186.

Supported Cryptography Classes

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

- `javacardx.crypto.AEADCipher`
- `javacardx.crypto.Cipher`
- `javacard.security.Checksum`
- `javacardx.security.derivation.DerivationFunction`
- `javacardx.security.cert.CertificateParser`
- `javacard.security.InitializedMessageDigest`
- `javacard.security.KeyAgreement`

- `javacard.security.KeyBuilder`
- `javacard.security.KeyPair`
- `javacard.security.MessageDigest`
- `javacard.security.RandomData`
- `javacard.security.Signature`
- `javacard.security.SignatureMessageRecovery`

ol lists the cryptography algorithms that are implemented for the simulator.

Table A-1 Algorithms Implemented by the Cryptography Classes

Class	Algorithm
`AEADCipher`	Supports `ALG_AES_CCM` and `ALG_AES_GCM` (supports only the 12 byte IV length, which is the value recommended by NIST)
`Checksum`	• `ALG_ISO3309_CRC16`—ISO/IEC 3309-compliant 16-bit CRC algorithm. This algorithm uses the generator polynomial: `x^16+x^12+x^5+1`. The default initial checksum value used by this algorithm is 0. This algorithm is also compliant with the frame-checking sequence as specified in section 4.2.5.2 of the ISO/IEC 13239 specification. • `ALG_ISO3309_CRC32`—ISO/IEC 3309-compliant 32-bit CRC algorithm. This algorithm uses the generator polynomial: `X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8 +X^7+X^5+X^4+X^2+X+1`. The default initial checksum value used by this algorithm is 0. This algorithm is also compliant with the frame-checking sequence as specified in section 4.2.5.3 of the ISO/IEC 13239 specification.

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

Class	Algorithm
Cipher	• ALG_DES_CBC_NOPAD —provides a cipher using DES in CBC mode without padding. This algorithm uses CBC for DES and 3DES. • ALG_DES_CBC_ISO9797_M1—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 1 scheme. • ALG_DES_CBC_ISO9797_M2—provides a cipher using DES in CBC mode. This algorithm uses EBC for DES and 3DES. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV'96) scheme. • ALG_DES_ECB_NOPAD —provides a cipher using DES in ECB mode without padding. This algorithm uses ECB for DES and 3DES. • ALG_DES_ECB_ISO9797_M1—provides a cipher using DES in ECB mode. This algorithm uses ECB for DES and 3DES. Input data is padded according to the ISO 9797 method 1 • ALG_DES_ECB_ISO9797_M2—provides a cipher using DES in ECB mode. This algorithm uses ECB for DES and 3DES. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV'96) scheme. • ALG_RSA_NOPAD - provides a cipher using RSA without padding. • ALG_RSA_PKCS1_OAEP - provides a cipher using RSA using the Optimal Asymmetric Encryption Padding scheme (OAEP). The following combinations (cipher algorithm, padding algorithm) are also supported: (CIPHER_RSA,PAD_PKCS1_OAEP), (CIPHER_RSA,PAD_PKCS1_OAEP_SHA256), (CIPHER_RSA,PAD_PKCS1_OAEP_EXT_PARAMETERS) The last combination also permits customization of the MGF1 digest algorithm with ALG_SHA or ALG_SHA256. • ALG_RSA_PKCS1—provides a cipher using RSA. Input data is padded according to the PKCS#1 (v1.5) scheme. • ALG_AES_BLOCK_128_CBC_NOPAD—provides a cipher using AES with block size 128 in CBC mode and does not pad input data. • ALG_AES_BLOCK_128_ECB_NOPAD—provides a cipher using AES with block size 128 in ECB mode and does not pad input data. • ALG_AES_CBC_ISO9797_M1—provides a cipher using AES with block size 128 in CBC mode. Input data is padded according to the ISO 9797 method 1 scheme. • ALG_AES_CBC_ISO9797_M2—provides a cipher using AES with block size 128 in CBC mode. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV’96) scheme. • ALG_AES_ECB_ISO9797_M1—provides a cipher using AES with block size 128 in ECB mode. Input data is padded according to the ISO 9797 method 1 scheme. • ALG_AES_ECB_ISO9797_M2—provides a cipher using AES with block size 128 in ECB mode. Input data is padded according to the ISO 9797 method 2 (ISO 7816-4, EMV’96) scheme. • ALG_AES_CBC_PKCS5—provides a cipher using AES with block size 128 in CBC mode. Input data is padded according to the PKCS#5 scheme. • ALG_AES_ECB_PKCS5—provides a cipher using AES with block size 128 in ECB mode. Input data is padded according to the PKCS#5 scheme.

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

Class	Algorithm
	• ALG_AES_XTS—provides a cipher using AES in XEX Tweakable Block Cipher with Ciphertext Stealing (XTS) mode as defined in IEEE Std 1619. AES keys of 128-bit and 256-bit length are supported. • ALG_AES_CFB—provides a cipher using AES in Cipher Feedback (CFB) mode. • ALG_KOREAN_SEED_ECB_NOPAD and ALG_KOREAN_SEED_CBC_NOPAD — provides a cipher using Korean Seed in ECB or CBC mode with a 128 bit long Korean Seed key
InitializedMessageDigest	Provides the functionality to create MessageDigest, with the additional ability to allow for initialization with a starting hash value corresponding to a previously hashed part of the message. Supports Message digest algorithms MD5, RIPEMD160, SHA1, SHA_224, SHA_256, SHA_384 and SHA_512.
KeyAgreement	• ALG_DH_PLAIN — Diffie-Hellman (DH) secret value deviration primitive as per NIST Special Publication 800-56Ar2. • ALG_EC_PACE_GM — elliptic curve Generic Mapping according to TR03110 v2. • ALG_EC_SVDP_DH_KDF — elliptic curve secret value derivation primitive, Diffie-Hellman version, per [IEEE P1363]. • ALG_EC_SVDP_DH_PLAIN_XY — elliptic curve secret value derivation primitive Diffie-Hellman version, as per [IEEE P1363]. • ALG_EC_SVDP_DH_PLAIN — elliptic curve secret value derivation primitive, Diffie-Hellman version, per [IEEE P1363]. • ALG_EC_SVDP_DHC_PLAIN — elliptic curve secret value derivation primitive, Diffie-Hellman version, with cofactor multiplication, and compatibility mode as per [IEEE P1363]. • ALG_EC_SVDP_DHC_KDF — elliptic curve secret value derivation primitive, Diffie-Hellman version, with cofactor multiplication, and compatibility mode as per [IEEE P1363].
KeyBuilder	Provide the functionality to create the following supported key types and key length: • 128, 192, 256 -bit ALG_TYPE_AES and 512-bit only for AES-XTS • 64, 128 and 192-bit ALG_TYPE_DES • 112-, up to 521-bit ALG_TYPE_EC_FP • 113-, up to 193-bit ALG_TYPE_EC_F2M • 512-, up to 4096-bit for ALG_TYPE_RSA and ALG_TYPE_RSA_CRT_PRIVATE • 8-, up to 512-bit ALG_TYPE_HMAC • 8-, up to 1024-bit ALG_TYPE_GENERIC_SECRET • 1024-bit and 2048-bit ALG_TYPE_DH • 1024-bit and 2048-bit ALG_TYPE_DSA
KeyPair	Provides the functionality to create the following supported key pairs types and length: • 112-, 128-, 160-, 192-, 224-, 256-, 384-, 521-bit ALG_EC_FP • 113-, 131-, 163-, 193-bit ALG_EC_F2M • 512-, 736-, 768-, 869-, 1024-, 1280-, 1536-, 2048-, 3072-, 4096-bit ALG_RSA and ALG_RSA_CRT • 1024-bit and 2048-bit ALG_DH • 1024-bit and 2048-bit ALG_DSA

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

Class	Algorithm
MessageDigest	Message digest supported algorithm: ALG_MD5, ALG_RIPEMD160, ALG_SHA1, ALG_SHA_224, ALG_SHA_256, ALG_SHA_384 and ALG_SHA_512.
RandomData	Supported random byte generation algorithms: ALG_FAST, ALG_KEYGENERATION, ALG_PRESEEDED_DRBG, ALG_TRNG, ALG_PSEUDO_RANDOM and ALG_SECURE_RANDOM

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

Class	Algorithm
Signature	• ALG_DES_MAC8_NOPAD—generates an 8-byte MAC (most significant 8 bytes of encrypted block) using DES or 3DES in CBC mode with no padding applied. The following combination of (cipher algorithm, digest algorithm, padding algorithm) is also supported: (SIG_CIPHER_DES_MAC8,ALG_NULL,PAD_NOPAD) • 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. The following combination of (cipher algorithm, digest algorithm, padding algorithm) is also supported: (SIG_CIPHER_DES_MAC8,ALG_NULL,PAD_ISO9797_M2) • ALG_DES_MAC8_ISO9797_1_M2_ALG3— generates an 8-byte MAC using triple DES with 2 keys. The MAC is according to "MAC algorithm 3" of ISO 9797-1 with padding according to method 2 (see also EMV 3.0 and EMV 4.0). The following combination of (cipher algorithm, digest algorithm, padding algorithm) is also supported: (SIG_CIPHER_DES_MAC8,ALG_NULL,PAD_ISO9797_M2) • ALG_DES_MAC8_PKCS5—generates an 8-byte MAC (most significant 8 bytesof encrypted block) using DES or 3DES in CBC mode.This algorithm uses CBC for DES and 3DES. Input data is padded according to PKCS#5. The following combination of (cipher algorithm, digest algorithm, padding algorithm) is also supported: (SIG_CIPHER_DES_MAC8,ALG_NULL,PAD_PKCS5) • ALG_DSA_SHA, ALG_DSA_SHA224, ALG_DSA_SHA256 signs or verifies a message that is hashed with SHA1, SHA_224 or SHA-256. The following combinations (cipher algorithm, digest algorithm, padding algorithm) are also supported: (SIG_CIPHER_DSA,ALG_SHA,PAD_NULL), (SIG_CIPHER_DSA,ALG_SHA_224,PAD_NULL), (SIG_CIPHER_DSA,ALG_SHA_256,PAD_NULL) • ALG_RSA_MD5_PKCS1, ALG_RSA_RIPEMD160_PKCS1—encrypts the MD5 or the RIPEMD160 message digest using RSA. The digest is padded according to the PKCS#1(v1.5) scheme. The following combinations (cipher algorithm, digest algorithm, padding algorithm) are also supported: (SIG_CIPHER_RSA,ALG_MD5,PAD_NULL), (SIG_CIPHER_RSA,ALG_RIPEMD160,PAD_NULL • ALG_RSA_SHA_PKCS1, ALG_RSA_SHA_224_PKCS1, ALG_RSA_SHA_256_PKCS1, ALG_RSA_SHA_384_PKCS1, ALG_RSA_SHA_512_PKCS1— Providing a Signature using the RSA algorithm for signing or verifying a message . The message digest is padded according to the PKCS#1(v1.5) scheme. The following combinations of (cipher algorithm, digest algorithm, padding algorithm) are also supported: (SIG_CIPHER_RSA,ALG_SHA,PAD_PKCS1), (SIG_CIPHER_RSA, ALG_SHA_224, PAD_PKCS1), (SIG_CIPHER_RSA,ALG_SHA_256, PAD_PKCS1), (SIG_CIPHER_RSA,ALG_SHA_384, PAD_PKCS1), (SIG_CIPHER_RSA,ALG_SHA_512, PAD_PKCS1)

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

Class	Algorithm
	• ALG_RSA_SHA_PKCS1_PSS, ALG_RSA_SHA_224_PKCS1_PSS, ALG_RSA_SHA_256_PKCS1_PSS, ALG_RSA_SHA_384_PKCS1_PSS, ALG_RSA_SHA_512_PKCS1_PSS— provides a Signature using the Probabilistic Signature Scheme (PSS) for signing or verifying a message. The following combinations of (cipher algorithm, digest algorithm, padding algorithm) are also supported: (SIG_CIPHER_RSA,ALG_SHA,PAD_PKCS1_PSS), (SIG_CIPHER_RSA, ALG_SHA224, PAD_PKCS1_PSS), (SIG_CIPHER_RSA, ALG_SHA256, PAD_PKCS1_PSS), (SIG_CIPHER_RSA, ALG_SHA384, PAD_PKCS1_PSS), (SIG_CIPHER_RSA, ALG_SHA512, PAD_PKCS1_PSS), (SIG_CIPHER_RSA,ALG_SHA, PAD_PKCS1_PSS_EXT_PARAMETERS), (SIG_CIPHER_RSA, ALG_SHA224, PAD_PCKS1_PSS_EXT_PARAMETERS), (SIG_CIPHER_RSA, ALG_SHA256, PAD_PCKS1_PSS_EXT_PARAMETERS), (SIG_CIPHER_RSA, ALG_SHA384, PAD_PCKS1_PSS_EXT_PARAMETERS), (SIG_CIPHER_RSA, ALG_SHA512, PAD_PCKS1_PSS_EXT_PARAMETERS). The last five combinations permit also to customize the mask generation function (MGF1) digest algorithm with ALG_SHA or ALG_SHA256. • ALG_AES_MAC_128_NOPAD—generates a 16-byte MAC using AES with block size 128 in CBC mode and does not pad input data. The following combination of (cipher algorithm, digest algorithm, padding algorithm) is also supported: (SIG_CIPHER_AES_MAC128,ALG_NULL,PAD_NOPAD) • ALG_AES_CMAC_128 —generates a 16-byte Cipher base MAC using AES with block size 128 in CBC with ISO9797-M2 padding. The following combination of (cipher algorithm, digest algorithm, padding algorithm) is also supported: (SIG_CIPHER_AES_CMAC128,ALG_NULL,PAD_ISO9797_M2) • ALG_ECDSA_SHA, ALG_ECDSA_SHA_224, ALG_ECDSA_SHA_256, ALG_ECDSA_SHA_384, ALG_ECDSA_SHA_512 — Provides a Signature using ECDSA for signing or verifying a message. . The following combinations of (cipher algorithm, digest algorithm, padding algorithm) are also supported with the signatures as ASN.1 SEQUENCE: (SIG_CIPHER_ECDSA,ALG_SHA, PAD_NULL), (SIG_CIPHER_ECDSA,ALG_SHA_224, PAD_NULL), (SIG_CIPHER_ECDSA,ALG_SHA_256, PAD_NULL), (SIG_CIPHER_ECDSA,ALG_SHA_384, PAD_NULL), (SIG_CIPHER_ECDSA,ALG_SHA_512, PAD_NULL) And the following combination for a ECDSA signature encoded as octet string: (SIG_CIPHER_ECDSA_PLAIN,ALG_SHA, PAD_NULL), (SIG_CIPHER_ECDSA_PLAIN,ALG_SHA_224, PAD_NULL), ,(SIG_CIPHER_ECDSA_PLAIN,ALG_SHA_256, PAD_NULL),

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

Class	Algorithm
	(SIG_CIPHER_ECDSA_PLAIN,ALG_SHA_384, PAD_NULL), (SIG_CIPHER_ECDSA_PLAIN,ALG_SHA_512, PAD_NULL) • ALG_HMAC_SHA1 and ALG_HMAC_SHA_256— generates an HMAC using the steps found in RFC 2104 using the SHA1 and SHA-256 standards, respectively as the hashing algorithm. The following combinations of (cipher algorithm, digest algorithm, padding algorithm) are also supported: (SIG_CIPHER_HMAC,ALG_SHA,PAD_NULL), (SIG_CIPHER_HMAC,ALG_SHA_256, PAD_NULL)
SignatureMessageRecovery	• ALG_RSA_SHA_ISO9796_MR or (CIPHER_RSA, ALG_SHA, PAD_ISO9796_MR) — RSA ISO9796 signature with message recovery scheme1 trailer field option1 and SHA1 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. The following combinations of (cipher algorithm, digest algorithm, padding algorithm) is also supported: (SIG_CIPHER_RSA,ALG_SHA,PAD_ISO9796_MR) • The following combinations of (cipher algorithm, digest algorithm, padding algorithm) (SIG_CIPHER_RSA, ALG_SHA, PAD_ISO9796_MR_SCHEME_1_OPT_2) RSA ISO9796 signature with message recovery scheme1 trailer field option2 and SHA1 algorithm is supported.
DerivationFunction	• ALG_HKDF_EXPAND_LABEL_TLS13 — Algorithm implementing the HKDF Expand Label version 1.3 for both TLS 1.3 (RFC 8446) and DTLS 1.3 (RFC 9147) cases. • ALG_KDF_HKDF — Algorithm implementing the HKDF Key Derivation function defined in IETF RFC 5869 with HMAC-SHA-1 or HMAC-SHA-256.
CertificateParser	• TYPE_X509_DER—parser for X.509 v1, v2, and v3 DER-encoded certificates (see RFC 5280). • ALG_RSA_SHA_PKCS1—is the supported signature algorithm.

Instantiating the Classes

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

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

If you request an algorithm that is not listed in Supported Cryptography Classes or that is not implemented in this release, getInstance throws a CryptoException with reason code NO_SUCH_ALGORITHM.

B

Annex - Application Management

A subset of GlobalPlatform Card Specification 2.3.1 [GPCS] is implemented by the simulator to support application management.

More precisely the simulator is implementing the subset defined in the "GlobalPlatform IoT Configuration 1.0". This includes the loading of CAP files, creating and deleting of applet instances, and deleting of packages. The following section describes the GPCS features that the simulator implements.

Referto the following publications, for more information about GlobalPlaform specifications:

- GlobalPlatform Card Specification v2.3.1, 2018, https://globalplatform.org
- GlobaPlatform Secure Channel Protocol '03' – Amendment D v1.2 https://globalplatform.org
- GlobalPlatform Card API (org.globalplatform) v1.6 https://globalplatform.org

Mapping of GlobalPlatform terminology to Java Card Simulator terminology:

Table B-1 Mapping terminology

GlobalPlatform	Java Card
Card Content Management	Application Management
Executable Load File (ELF)	CAP File
Executable Module	Package
Application	Applet

Before you can run the Simulatr with the GP functionality, ensure that the initial ISD keys and the Global PIN are configured correctly using the Configurator tool. Refer to the Installation section for additional information and instructions.

The Issuer Identification Number (IIN) and Card Image Number (CIN) are set to default values but can be configured through the command line when launching the Simulator. For instructions, refer to the Java Card Development Kit Simulator Command Line section.

Supported GlobalPlatform Features

The Issuer Security Domain (ISD) with the default AID defined by (GPCS) is the default selected applet after the start of the Simulator.

The ISD implements the secure channel protocol SCP03 with `option i='70'` specified in GP Amendment D. This implementation supports initial AES keys of length 128, 192 or 256 bit length. The keys cannot be updated. The initial keyset can be configured with the Configurator tool, please refer to the Installation section of this document for the configuration of the keyset.

> **Note:**
>
> The Key Version Number (KVN) for the initial ISD keys is set to '0x10'.

All the card lifecycle states specified in (GPCS) are supported. The Simulator is in lifecycle state OP_READY after the installation of the Simulator. It is possible to change the lifecycle state according to the rules defined in (GPCS). The ISD is in lifecycle state PERSONALIZED after the installation of the Simulator.

The ISD supports the following APDU commands. Please refer to the (GPCS) specification for the details of the command.

Supported APDU Commands

DELETE

The `DELETE` command is used to delete applet instances and packages.

GET DATA

The `GET DATA` command is used to retrieve a single or a set of BER-TLV-coded objects. The ISD supports the following objects identified by their `TAG`:

- **TAG '42'**: Issuer Identification Number (IIN)
- **TAG '45'** Card Image Number (CIN)
- IIN and CIN can be configured with command line options when starting the Simulator.
- **TAG '66'** Card Data
- **TAG 'E0'** Key Information Template
- **TAG '67'** Card Capability Information
- **TAG 'C1'** Sequence Counter of the default Key Version Number
- **TAG 'CF'** Key derivation data tag
- **TAG '2F00'** List of applications

GET STATUS

The `GET STATUS` command is used to retrieve ISD, CAP file, package and applet status information.

INSTALL

The `INSTALL` command is used to initiate and perform the various steps of the Card Content management defined in the (GPCS). The following command data fields are supported, for the details of these command data field please refer to the (GPCS) specification:

- `INSTALL [for load]`
- `INSTALL [for install]`
- `INSTALL [for make selectable]`

- `INSTALL [for personalization]`
- `INSTALL [install and make selectable]`

LOAD

The `LOAD` command is used to load a CAP file (compact as well as extended format) into the Simulator. A CAP file is usually loaded with multiple `LOAD` commands with a sequential numbering as described in (GPCS).

MANAGE CHANNEL

This command is used to open and close logical channel, the Simulator supports 4 logical channels. The processing of the `MANAGE CHANNEL` command follows the rules described in the JCRE specification and in the (GPCS).

SELECT

The `SELECT` command is used for selecting an Applet or a Security Domain (SD). The processing of the `SELECT` command follows the rules described in the JCRE specification and in the (GPCS).

SET STATUS

The `SET STATUS` command can be used to manage the card lifecycle state or the application life cycle state.

STORE DATA

The `STORE DATA` command is used to transfer data to an Applet or the ISD. If an Applet wants to receive these data it has to use the mechanism described in the `GlobalPlatform` API. The implementation of the Simulator does not support key loading for the ISD with the `STORE DATA` command.

GlobalPlatform API

The Simulator implements the **GlobalPlatform** API v. 1.6 (`org.globalplatform`). The documentation and the export files are available on the GlobalPlatform website. Refer to the link at the beginning of this chapter. The entire package is available in the Simulator but not all services are enabled. The following interfaces are not implemented:

- GlobalService
- HTTPAdministration
- HTTPReportListener
- Authority
- SecureChannelx
- SecureChannelx2

Glossary

active applet instance

an applet instance that is selected on at least one of the logical channels.

AID (application identifier)

defined by ISO 7816, a string used to uniquely identify card applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 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 assigned for each package. In addition, a unique AID is assigned for each applet in the package. The package AID and the default AID for each applet defined in the package are specified in the `CAP` file. They are supplied to the converter when the `CAP` file is generated.

APDU

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

API

an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services.

applet

within the context of this document, a Java Card applet, which is the basic unit of selection, context, functionality, and security in Java Card technology.

applet application

an application that consists of one or more applets.

applet CAP file

a CAP file that contains one or more applet packages. See applet package.

applet developer

a person creating an applet using Java Card technology.

applet execution context

currently active applet owner identifier.

applet framework

an API that enables applet applications to be built.

applet firewall

the mechanism that prevents unauthorized accesses to objects in contexts other than currently active context.

applet package

a Java programming language package that contains one or more non-abstract classes that extend the `javacard.framework.Applet` class. See also library package.

assigned logical channel

the logical channel on which the applet instance is either the active applet instance or will become the active applet instance.

atomic operation

an operation that either completes in its entirety or no part of the operation completes at all.

atomicity

property in which a particular operation is atomic. Atomicity of data updates guarantees that data are not corrupted in case of power loss or card removal.

ATR

an acronym for Answer to Reset. An ATR is a string of bytes sent by the Java Card platform after a reset condition.

authentication

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

basic logical channel

logical channel 0, the only channel that is active at card reset in the APDU application environment. This channel is permanent and can never be closed.

big-endian

a technique of storing multibyte data where the high-order bytes come first. For example, given an 8-bit data item stored in big-endian order, the first bit read is considered the high bit.

binary compatibility

in a Java Card system, a change in a Java programming language package in a Java Card CAP file results in a new CAP file. A new CAP file is binary compatible with (equivalently, does not break compatibility with) a preexisting CAP file if another CAP file converted using the export files of the packages included in the preexisting CAP file can link with the new CAP file without errors.

bytecode

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

CAD

an acronym for Card Acceptance Device. The CAD is the device in which the card is inserted.

CAP file

Standard file format containing a binary representation of a shared library (library CAP file) or an application with its libraries that might be exported or not (applet CAP file).

A CAP file represents a module, which is a unit of code, made of one or more Java packages, with dependencies and list of exported packages and an assigned name (AID) for lifecycle management. Its structure is made of multiple CAP components deployed within a JAR file

When a CAP file containing application(s) is deployed on a Java Card platform, it is assigned a new unique group context that must be associated with any application instance created from code within this application module.

CAP file component

A Java Card platform CAP file consists of a set of components, which represent a set of one or more Java programming language packages. Each component describes a set of elements or an aspect of the CAP file. A complete CAP file must contain all of the required components: Header, Directory, Import, Constant Pool, Method, Static Field, and Reference Location.

The following components are conditionally included or optional: the Applet, Export, Static Resources and Debug. The Applet component is included only if one or more Applets are

defined in one or more packages in the CAP file. The Export component is included only if one or more packages are public and exported allowing classes in other packages to import elements from them. The Static Resources component is included only if static resources are embedded in the CAP file. The Debug component is optional. It contains all of the data necessary for debugging.

cast

the explicit conversion from one data type to another.

card session

a card session begins with the insertion of the card into the CAD. The card is then able to exchange streams of APDUs with the CAD. The card session ends when the card is removed from the CAD.

client application

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

constant pool

the constant pool contains variable-length structures representing various string constants, class names, field names, and other constants referred to within the CAP file and the Export File structure. Each of the constant pool entries, including entry zero, is a variable-length structure whose format is indicated by its first tag byte. There are no ordering constraints on entries in the constant pool. One constant pool is associated with each package.

There are differences between the Java platform constant pool and the Java Card technology-based constant pool. For example, in the Java platform constant pool there is one constant type for method references, while in the Java Card constant pool, there are three constant types for method references. The additional information provided by a constant type in Java Card technologies simplifies resolution of references.

context

protected object space associated with each applet `CAP` file and Java Card RE. All objects owned by an applet belong to the context associated with the applet's `CAP` file.

context switch

a change from one currently active context to another. For example, a context switch is caused by an attempt to access an object that belongs to an application instance that

resides in a different application group. The result of a context switch is a new currently active context.

Converter

a piece of software that preprocesses all of the Java programming language class files that make up a package, and converts the package to a `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.

currently selected applet

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

custom CAP file component

a new component added to the `CAP` file. The new component must conform to the general component format. It is silently ignored by a Java Card virtual machine that does not recognize the component. The identifiers associated with the new component are recorded in the `custom_component` item of the `CAP` file's Directory component.

default applet

an applet that is selected by default on a logical channel when it is opened. If an applet is designated the default applet on a particular logical channel on the Java Card platform, it becomes the active applet by default when that logical channel is opened using the basic channel.

EEPROM

an acronym for Electrically Erasable, Programmable Read Only Memory.

entry point objects

see Java Card RE entry point object.

Export file

a file produced by the Converter that represents the fields and methods of a package that can be imported by classes in other packages.

externally visible

in the Java Card platform, any classes, interfaces, their constructors, methods, and fields of an application that can be accessed from another application according to the Java programming language semantics, as defined by the *Java Language Specification*.

Externally visible items are represented in an export file. For a library package, externally visible items are represented in an export file. For an applet package, only those externally visible items that are part of a shareable interface are represented in an export file.

A Java Card `CAP` file may restrict the visibility of a package it contains. In this case, these packages are only visible to the other packages inside the `CAP` file and are not be accessible by packages in other `CAP` files. No export file is generated for the packages that have their visibility restricted to packages inside the same `CAP` file.

finalization

the process by which a Java virtual machine (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. There is no `finalize()` method to be called automatically by the Java Card virtual machine.

firewall

see applet firewall.

flash memory

a type of persistent mutable memory. It is more efficient in space and power than EEPROM. Flash memory can be read bit by bit but can be updated only as a block. Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole.

framework

the set of classes that implement the API. This includes core and extension packages. Responsibilities include applet selection, sending APDU bytes, and managing atomicity.

garbage collection

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

global array

an array objects accessible from any context.

group context

protected object space associated with each `CAP` file and Java Card RE defining the boundaries of the firewall.

heap

a common pool of free memory in volatile and persistent spaces usable by a program for dynamic memory allocation, in which blocks of memory are used in an arbitrary order. The Java Card virtual machine's heap is not required to be garbage collected and objects allocated from the heap are not necessarily reclaimed.

installation program

the off-card mechanism that employs a card acceptance device (CAD) to transmit the executable binary in a `CAP` file to the installer running on the card.

installer

the on-card mechanism to download and install `CAP` files. The installer receives executable binary from the off-card installation program, writes the binary into the smart card memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card Runtime Environment.

instance variables

also known as non-static fields.

instantiation

in object-oriented programming, to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and

initialization of instance variables with either default values or those provided by the class's constructor function.

instruction

a statement that indicates an operation for the computer to perform and any data to be used in performing the operation. An instruction can be in machine language or a programming language.

internally visible

code items that are not externally visible. These items are not described in a package's export file and use private tokens to represent internal references. See externally visible

JAR file

an acronym for Java Archive file, which is a file format used for aggregating and compressing many files into one.

Java Card Platform Remote Method Invocation

a subset of the Java Platform Remote Method Invocation (RMI) system. It provides a mechanism for a client application running on the CAD platform to invoke a method on a remote object on the card.

Java Card RE context

the context of the Java Card RE has special system privileges so that it can perform operations that are denied to contexts of applications.

Java Card RE entry point object

an object owned by the Java Card RE context that contains entry point methods. These methods can be invoked from any context and allows applications to request Java Card RE system services. A Java Card RE entry point object can be either temporary or permanent:

temporary - references to temporary Java Card RE entry point objects cannot be stored in class variables, instance variables or array components. The Java Card RE detects and restricts attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. Examples of these objects are APDU objects and the APDU byte array.

permanent - references to permanent Java Card RE entry point objects can be stored and freely reused. Examples of these objects are Java Card RE-owned AID instances.

Java Card Runtime Environment (Java Card RE)

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

Java Card Virtual Machine (Java Card VM)

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

JDK software

an acronym for Java Development Kit. The JDK software provides the environment required for software development in the Java programming language. The JDK software is available for a variety of operating systems.

library CAP file

a CAP file that contains only library packages. See library package.

library package

a Java programming language package that does not contain any non-abstract classes that extend the class `javacard.framework.Applet`. See also applet package.

local variable

a data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and cannot be used outside the method.

logical channel

as seen at the card edge, works as a logical link to an applet application on the card. A logical channel establishes a communications session between a card applet and the terminal. Commands issued on a specific logical channel are forwarded to the active applet on that logical channel. For more information, see the *ISO/IEC 7816 Specification, Part 4*. (`http://www.iso.org`).

MAC

an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.

mask production (masking)

refers to embedding the Java Card virtual machine, runtime environment, and applets in the read-only memory of a smart card during manufacture.

multiselectable applets

implements the `javacard.framework.MultiSelectable` interface. Multiselectable applets can be selected on multiple logical channels at the same time. They can also accept other applets belonging to the same package being selected simultaneously.

multiselected applet

an applet instance that is selected and, therefore, active on more than one logical channel simultaneously.

namespace

a set of names in which all names are unique.

native method

a method that is not implemented in the Java programming language, but in another language. The `CAP` file format does not support native methods.

nibble

four bits.

non-volatile memory

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

object

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.

object-oriented

a programming methodology based on the concept of an *object*, which is a data structure encapsulated with a set of routines, called *methods*, which operate on the data.

origin logical channel

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

owner context

see owning context.

owning context

the application or Java Card RE context in which an object is instantiated or created.

package

a namespace within the Java programming language that can have classes and interfaces.

PCD

an acronym for Proximity Coupling Device. The PCD is a contactless card reader device.

persistent object

persistent objects and their values persist from one CAD session to the next, indefinitely. Objects are persistent by default. Persistent object values are updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized and deserialized, just that the objects are not lost when the card loses power.

PIX

see AID (application identifier).

RAM (random access memory)

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

reference implementation

functional and fully compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.

remote interface

an interface of an applet application, which extends, directly or indirectly, the interface `java.rmi.Remote`.

Each method declaration in the remote interface or its super-interfaces includes the exception `java.rmi.RemoteException` (or one of its superclasses) in its `throws` clause.

In a remote method declaration, if a remote object is declared as a return type, it is declared as the remote interface, not the implementation class of that interface.

In addition, Java Card RMI imposes additional constraints on the definition of remote methods of an applet application. See *Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.2*.

remote methods

the methods of a remote interface of an applet application.

remote object

an object of an applet application whose remote methods can be invoked remotely from the off-card client. A remote object is described by one or more remote interfaces of an applet application.

RFU

acronym for Reserved for Future Use.

RID

see AID (application identifier).

RMI

an acronym for Remote Method Invocation. RMI is a mechanism for invoking instance methods on objects located on remote virtual machines (meaning, a virtual machine other than that of the invoker).

ROM (read-only memory)

memory used for storing the fixed program of the card. A smart card's ROM contains operating system routines as well as permanent data and user applications. No power is needed to hold data in this kind of memory. ROM cannot be written to after the card is manufactured. Writing a binary image to the ROM is called masking and occurs during the chip manufacturing process.

runtime environment

see Java Card Runtime Environment (Java Card RE).

service

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

shareable interface

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

shareable interface object (SIO)

an object that implements the shareable interface.

smart card

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

SPI

an acronym for Service Provider Interface or sometimes for System Programming Interface. The SPI defines calling conventions by which a platform implementer may implement system services.

terminal

is typically a computer in its own right with an interface which connects with a smart card to exchange and process data.

thread

the basic unit of program execution. A process can have several threads running concurrently each performing a different job, such as waiting for events or performing a time consuming job that the program doesn't need to complete before going on. When a thread has finished its job, it is suspended or destroyed.

The Java Card virtual machine can support only a single thread of execution. Java Card technology programs cannot use class Thread or any of the thread-related keywords in the Java programming language.

transaction

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

transient object

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

uniform resource identifier (URI)

a compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both. See RFC 3986 for more information.

uniform resource locator (URL)

a compact string representation used to locate resources available via network protocols or other protocols. Once the resource represented by a URL has been accessed, various operations may be performed on that resource. See RFC 1738 for more information. A URL is a type of uniform resource identifier (URI).

verification

a process performed on an application or library executable that checks that the binary representation of the application or library is structurally correct and type safe.

volatile memory

memory that is not expected to retain its contents between card tear and power up events or across a reset event on the smart card device.

volatile object

an object that is ideally suited to be stored in volatile memory. This type of object is intended for a short-lived object or an object which requires frequent updates. A volatile object is garbage collected on card tear (or reset).

word

an abstract storage unit. A word is large enough to hold a value of type `byte`, `short`, `reference` or `returnAddress`. Two words are large enough to hold a value of `integer` type.

Index