Java Card
Development Kit Tools User Guide

Version 24.0
F91373-02

Java Card Development Kit Tools User Guide, Version 24.0

F91373-02

Contents

Glossary

List of Figures

List of Tables

Preface

This document describes how to use the Java Card Development Kit Tools, to convert and verify 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.

> **Note:**
>
> The Java Card Development Kit Tools is released in both binary and source bundles. The Java Card Development Kit Tools binary bundle is publicly available on OTN. Access to the source bundles requires the purchase of a commercial license from Oracle. Besides, some source bundles may not include cryptography extensions, which are subjected to export restrictions.

Audience

This *Development Kit Tools User Guide* is written for developers who are creating applets using the *Application Programming Interface, Java Card Platform, Version 3.2, 3.1, 3.0.5, 3.0.4* 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/corporate/accessibility/.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit https://support.oracle.com/portal or visit https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab 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*
- *Java Card RMI Client Application Programming Interface* (see the Javadoc tool generated API specification at *JC_HOME_TOOLS*`\docs\rmiclientlib`)
- *ISO 7816-4:2013 Specification*

Documentation and Support

These web sites provide additional resources:

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

Third-Party Web Sites

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

Conventions

The following text conventions are used in this document:

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

Part I
Setup and Use of Tools

This part of the User Guide describes how to install the Development Kit and use the Tools. It contains the following chapters:

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 Specifications.

The complete development kit 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 should 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 testing and debugging reference for Java Card applications. It includes a Java Card simulation environment and it can be used from command line or from the Eclipse Plug-in. It provides support for the latest Java Card 3.2 Specification, and can also run applications written for earlier releases.
- The Java Card Development Kit Eclipse Plug-in offers the GUI for developing Java Card applets and interacting with Java Card Development Kit Tools and Simulator.

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 Tools bundle. The Java Card Development Kit Simulator and Java Card Development Kit Eclipse Plug-in bundles have their own dedicated User Guide. For information, refer to the Installing the Development Kit Simulator and Java Card Eclipse Plug-in .

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

> **Note:**
>
> The Java Card Development Kit Simulator is only designed as an example of the functional behavior of a Java Card runtime. It is not intended to operate in a production environment (and under the threats typically associated with such an environment).

2
Installation

This chapter describes the software that you must install on your system before you can use the development kit tools, how to install the development kit tools, how to check system variables, and how to uninstall the development kit tools.

This section only applies to the binary bundles of the Java Card Development Kit Simulator and the Java Card Development Kit Tools.

The development kit is available as independent downloads:

- The Java Card Development Kit Tools are used to convert and verify Java Card applications. The Tools can be used with products based on versions 3.2, 3.1, 3.0.5, and 3.0.4 of the Java Card Specifications.
- The Java Card Development Kit Simulator offers a testing and debugging reference for Java Card applications. It includes a Java Card simulation environment that can be run from the command line or from the Eclipse Plug-in. It provides support for the latest Java Card 3.2 Specification, and can also run applications written for earlier releases.
- The Java Card Development Kit Eclipse Plug-in offers the GUI for developing Java Card applets and interacting with Java Card Development Kit Tools and Simulator.

This chapter contains the following sections:

Install and Setup the Java Card Development Kit Tools

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

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

Before Installing the Java Card Development Kit Tools

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

- **Java Development Kit (JDK)** - The tools in this development kit were tested with Oracle JDK 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

https://jdk.java.net/archive/

- **Apache Ant (Optional)** - Version 1.10 was used to test the release. You can download and install Apache Ant from https://ant.apache.org.

Installing the Java Card Development Kit Tools

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

The Java Card Development Kit Tools is available for download at https://www.oracle.com/java/java-card/ .

1. Download the Java Card Development Kit Tools `.zip` file to a directory of your choice.
   - `java_card_devkit_tools-<ea>-bin-v[version]-<buildID>-<dd-mmm-yyy>.zip`
2. Extract the downloaded `.zip` file to the directory of your choice.

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

Confirming System Variables

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

- *JAVA_HOME* system variable must be set to the JDK software root directory and its `bin` in the `PATH`.
- *ANT_HOME* system variable must be set to the Ant root directory and its `bin` in the `PATH`.
- *JC_HOME_TOOLS* variable must be set to the Java Card Development Kit Tools root directory.
- The Java Card development kit `bin` directory must be in the `PATH` .

Installed Files and Directories

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

Uninstalling the Java Card Development Kit Tools

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

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

3
Developing Java Card Applications

This chapter provides an introduction to developing Java Card applications. You should also refer to the *Application Programming Interface, Java Card Platform, Classic Edition, Version 3.2* for additional information.

This chapter contains the following sections:

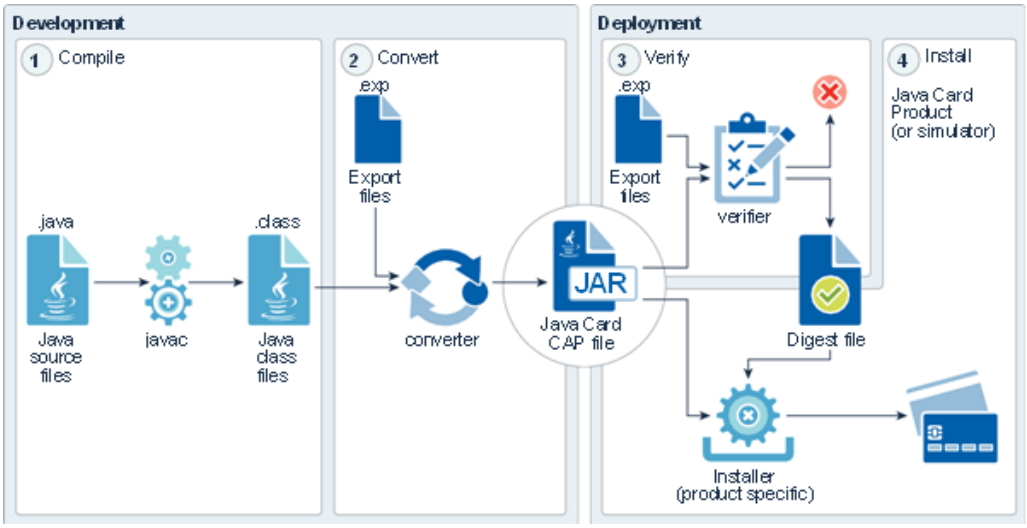
Java Card Applet Development

To develop an applet, you should do the following:

- **Install and Setup** — Install and setup the development environment. See Installation.
- **Review Samples** — Read, run the samples, and examine the code from the Simulator bundles.
- **Develop** — Develop your applet and compile the code to create the Java class files. Then use the Java Card Development Kit Tools to convert the classes and create a CAP file that can be deployed into the simulator. See Using Java Card Development Kit Tools for more information on how to use the tools.
- **Deploy** — Deploy your application to the Java Card simulator.
- **Debug** — Debug the applet. Use the Java Card debug proxy included in the Simulator development kit.

The figure shows the applet development and deployment process.

Figure 3-1 Process for Applet Development and Deployment



Java Card Development Kit Components

The Development Kit is available as three independent downloads.

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

Java Card Development Kit Eclipse Plug-in - The plug-in provides a way to run the tools in this list from inside Eclipse.

Java Card Development Kit Tools - Used to convert and verify Java Card appliations. The Tools can be used with products based on version 3.0.4, 3.0.5, 3.1, and 3.2 of the Java Card Specifications.

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

Using Java Card Development Kit Tools

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

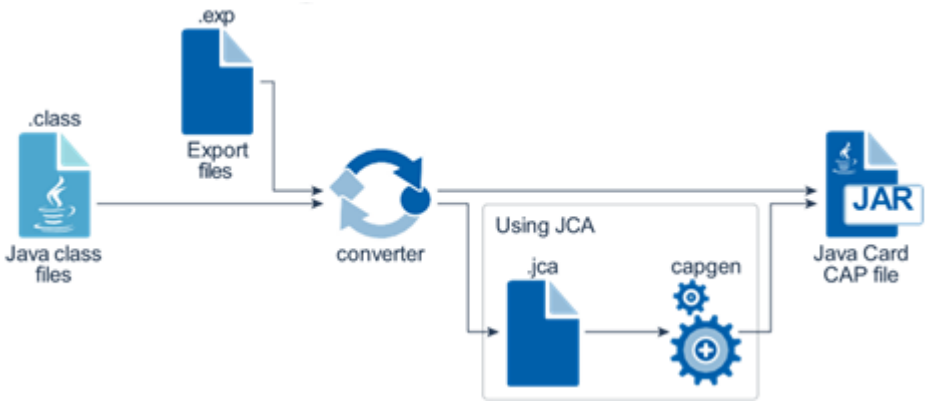
A Java Card CAP file is a JAR file containing the binary representation of a unit of code, made of one or more Java packages. It can be distributed for deployment on real devices running Java Card or simulators.

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

To deploy the CAP files on the Java Card simulator compliant with Global Platform specifications, use the Application Management API, implementing related Global Platform commands, included in the Java Card Development Kit Simulator.

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

Figure 3-2 Using Java Card Assembly



4

Converting and Exporting Java Class Files

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

Overview of Converting and Exporting Java Class Files

The Converter processes all of the Java class files that make up an application (or a library) and creates a binary file (CAP file) that can be deployed and loaded on a Java Card platform. The class files provided as input to the converter must have the class file major version between 45 (JDK 1.0) and 51 (Java™ SE 7), and it is recommended to use the Java SE 17 compiler to generate them. It also produces other files (export files and JCA files) that are used in the development and deployment process. The CAP file contains a manifest file that provides human-readable information about its content. See Working With CAP Files and Using Export Files, for more information.

Since Java Card Platform Specification, Version 3.1, the CAP file supports the following formats:

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

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

- See the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.2* for more information on the CAP file and its format.
- See Chapter Programming Large Java Card Applications with Multiple Packages for more information on using the Extended CAP file format.

Using the Converter in the Compact or Extended Format

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

- **Application Design**

  - The number of Java packages included in the application - If the application (or library) only includes a single package, it can be converted into a compact file format.
  - The total code size - If the application code creates a method component with a size larger than 64 K, then the extended format is required.

- **Deployment Constraints**
  - Some libraries must remain private - If an application relies on libraries that must not be shared, the extended format can be used to support application made of multiple packages instead of refactoring the code and copying the library classes into the application package.
  - A shared library includes both a public API made of one or more exported packages and private implementation packages - This could be achieved using an extended format CAP file that contains both parts, keeping the implementation packages private, and deploying all of the packages in one CAP file.

The format (compact or extended) can be set using the Converter command line parameters. See Running the Converter, for more details about the command line parameters. See Programming Large Java Card Applications with Multiple Packages, for more details about the Extended CAP file.

Using the Converter for a Target Java Card Version

The Converter can be used to create CAP files for Java Card versions 3.0.4, 3.0.5, 3.1.0 and 3.2.0 using the `-target [version]` command line parameter.

If the CAP file needs to be deployed on multiple Java Card versions, use the oldest version (that is, the smallest version number) as the target.

Table 4-1 Converter Usage

API Version	Converter Usage	When to Use?	CAP File Format Generated
3.0.4	`-target 3.0.4`	Use this mode when the target platform is 3.0.4.	2.2 (compact)
3.0.5	`-target 3.0.5`	Use this mode when the target platform is 3.0.5.	2.2 (compact)
3.1.0	`-target 3.1.0`	Use this format when the target platform is 3.1.0 and the code size is less than 64 K, in one package.	2.3 (compact)
3.2.0	`-target 3.2.0 (default)`	Use this format when the target platform is 3.2.0 and the code size is less than 64 K in one package.	2.3 (compact)

Using the Converter to Generate a Mask

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

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

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

Setting Java Compiler Options

To set Java compiler options:

1. When using the recommended compiler (Java 17) or any Java compiler with a version great than 7, it is mandatory to use the `"-source 7"` and `"-target 7"` command line options.
2. Compile your class files with the Java Development Kit compiler's `-g` command line option.

The `"-source 7"` and `"-target 7"` command line options force the compiler to generate classes with class major version 51, which is the maximum and recommended class version for the Converter input class files.

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

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

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

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

Running the Converter

To run the Converter (use the `.sh` extension for the shell scripts file in Unix/Linux-like systems and use the `.bat` extension for batch script files in Windows systems):

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

`converter.[sh | bat]` [*options*] *package-name package-AID major-version*`.`*minor-version*

Or

`converter.[sh | bat] -config` <*filename*>

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

   `converter.[sh | bat] -config` <*filename.json*>

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

   `converter.[sh | bat] -help`

   Or

   `converter.[sh | bat] -help JSON`

> **Note:**
>
> The `converter.[sh | bat]` file used to invoke the actual Converter directly from your working directory is a batch/script file only. If you have added *JC_HOME_TOOLS*`\bin` to your `PATH` you can call it directly, if not, you must precede it with *JC_HOME_TOOLS*`\bin`.

The Converter command line options described in the table below allow you to:

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

  When a package is used as a mask, restrictions on native methods are relaxed.
- Specify support for the 32-bit integer type.
- Enable generation of debugging information.
- Turn off verification (the default of input and output files. Verification is the default.).
- Specify a list of file paths from where the static resources are loaded by the Converter, if any.

- Specify the target Java Card platform version on which the CAP file generated should be loaded, if it is not the newest released version of the Java Card platform.

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

1. **Loads the packages** - If the `exportmap` option is set for any of the packages, the Converter loads that package from the export path (see Specifying an Export Map). It loads the class files of the Java packages and creates a data structures to represent these packages.
2. **Subset checking** - Checks for unsupported Java features in class files.
3. **Conversion** - Checks for consistency between the applet AIDs, package AIDs, CAP file AID (if present), and the imported package AIDs.
4. **Reference Checking** - Checks that all references are valid, internal referenced items are defined in the packages belonging to the CAP file, and import items are declared in the export files (see Using Export Files).

   The Converter creates the `JcImportTokenTable` to store tokens for import items (class, methods, and fields). If the Converter only generates export files, it does not check private APIs and byte code. Also included is a second round of subset checking that operations do not exceed the limitations set by the JCVM specification.
5. **Optimization** - Optimizes the byte code.
6. **Generates output** - Builds and outputs one EXP export file for each package and one JCA file for each package, checks for each package version in the export file against the version specified in the command line or in the `config` file. If the `-exportmap` option is used for a specific package in the command line or config file, the export file specified in the command line for that package must represent the same version as that of the package. The converter does not support upgrading the export file version.

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

   [*root_dir/package_dir*`/javacard` | *root_dir**package_dir*`\javacard`]

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

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

   [*output_dir/CAP_name*`/javacard` | *output_dir**CAP_name*`\javacard`]

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

Table 4-2 Converter Command Line Arguments

Option	Description
`-help`	Prints help message.

Table 4-2 (Cont.) Converter Command Line Arguments

Option	Description
`-help JSON`	Prints a JSON definition file (schema), for the JSON configuration file to be used in extended mode. The JSON schema contains all of the fields that can be defined, the hierarchy of fields, field types, field descriptions, optionality, sample values, default values, and descriptions. The schema can be used (using various tools) for validating configuration files used for generating extended CAP files.
package-name	Fully-qualified name of the package to convert.
package-AID	5 to 16 decimal, hex or octal numbers separated by colons. Each of the numbers must be byte-length.
major-version minor-version	User-defined version of the package.
`-applet` *AID class_name*	Sets the default applet AID and the name of the class that defines the applet. If the package contains multiple applet classes, this option must be specified for each class.
`-classdir` *root-directory-of-class hierarchy*	Sets the root directory where the Converter looks for classes. If this option is not specified, the Converter uses the current user directory as the root.
`-d` *root-directory-for-output*	Sets the root directory for output.
`-debug`	Generates the optional debug component of a CAP file. If the `-mask` option is also specified, the file `debug.msk` is generated in the output directory. **Note:** To generate the debug component, you must first compile your class files with the Java compiler's `-g` option.
`-exportmap`	Uses the token mapping from the pre-defined export file of the package being converted. The Converter looks for the export file in the `exportpath`.
`-exportpath` *list-of-directories*	Specifies the root directories in which the Converter looks for export files. The separator character for multiple paths is the colon (`:`) for Unix/Linus OS or the semicolon (`;`) for Windows OS. If this option is not specified, the Converter sets the export path to the Java `classpath`.
`-i`	Instructs the Converter to support the 32-bit integer type.
`-mask`	Indicates that the converted code is intended to be used to create a binary mask, so restrictions on native methods are relaxed. If you have a source release, you can specify this option to generate a mask out of this package using `maskgen`. This option can be used in conjunction with `-out CAP`, only if `-debug` is selected, to typically generate a CAP with debug component and use it to debug platform classes. Such CAP is not intended to be loaded on a platform and will fail verification if it contains native methods.
`-nobanner`	Suppresses all banner messages.
`-noverify`	Suppresses the verification of input and output files. For more information on file verification, see Verification of Input and Output Files.
`-nowarn`	Instructs the Converter not to report warning messages.

Table 4-2 (Cont.) Converter Command Line Arguments

Option	Description
`-out` [*CAP*] [*EXP*] [*JCA*]	Instructs the Converter to output the CAP file, and/or the export file, and/or the Java Card Assembly file. By default (if this option is not specified), the Converter outputs a CAP file and an export file.
`-v`, `-verbose`	Enables verbose output. Verbose output includes progress messages, such as "opening file", "closing file", and whether the package requires integer data type support.
`-V`, `-version`	Prints the Converter version string.
`-sign`	Specifies to sign the output CAP file.
`-keystore` *value*	Keystore to use in signing.
`-storepass` *value*	Keystore password.
`-alias` *value*	Keystore alias to use in signing.
`-passkey` *value*	Alias password.
`-useproxyclass`	Cannot be specified with `keepproxysource`. Builds CAP files as usual in the specified output directory using the existing class files of the application and existing class files of the associated proxy sub-package. New proxy classes are not created. Provides a way for the application developer to build a CAP file with customized proxy files. This option requests the converter to take the class files of the application package and the class files of the co-located proxy sub-package to build a new CAP file. The classes in the application package are converted into new `.cap` components. New descriptors are created. Dynamically-loaded-classes attributes need to be recomputed based on the new Proxy class file names.
`-usecapcomponents`	Specifies that the converter retain the specified user supplied CAP components instead of generating them in the final CAP bundle. The input format is as follows: *application-classes-dir*`/`*application-classes*`/javacard/*.cap`
`-keepproxysource` *directory*	Cannot be used with `-useproxyclass`. Creates the proxy source files and other stub files in the specified *directory*. The converter also builds CAP files as usual in the specified output directory. Supports customizing the proxy files generated by the converter. Requests the converter retain the intermediate proxy class source code in the specified directory and the source code of the associated stub classes representing the dependent external classes using the hierarchical directory structure of the Java package name(s).
`-resourcepath` *<id1>*`:`*<resource_path1>*`,`*<id2>*`:`*<resource_path2>*`,...`	Specifies the list of static resources that can be loaded into the CAP file that is generated by the Converter (in the compact mode). The entries in the list are delimited by the "`,`" character. Each entry in the list contains two parameters delimited by the "`:`" character. The first parameter is an integer representing the *id* of the static resource and the second parameter is the *path* to the file, which has the actual binary content for the static resource. The *path* must be a valid path to a file on the disk for which the Converter should have read access.

Table 4-2 (Cont.) Converter Command Line Arguments

Option	Description
`-target` <*platform version*>	Specifies the Java Card platform version on which the CAP file that is generated by the Converter (in the compact mode) is loaded. If `-target` is not specified, the default value would be the latest release version, that is, 3.2.0. Other valid values for `-target` are 3.0.4, 3.0.5, and 3.1.0. If you omit the `-target` option or if you are using a target greater than 3.0.5, the 2.3 version CAP files are generated. Otherwise, 2.2 or 2.1 version CAP files are generated, depending on the used features (debugging or RMI). Additionally, for the current release, the platform's `api_export_files` directory is not required to be given in the `-exportpath` option, it is chosen automatically based on the `-target` option.

Using Delimiters with Command Line Options

To use delimiters with command line options:

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

In the following sample command line, the Converter checks for export files in both the `.\export files` and current directory.

```
converter.[sh|bat] -target 3.0.5 -exportpath [“./export files”:. |
“.\export files”;.] MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

Using a Command Configuration File in Compact Mode

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

To use a command configuration file:

1. Enter the command line arguments and options in a text-format configuration file.
2. Use double quote (") delimiters for the command line options that require arguments in the configuration file.

   You must use double quote (") delimiters for the command line options that require arguments in the configuration file. For example, if the options from the command line example used in Using Delimiters with Command Line Options were placed in a configuration file, the result would look like this:

   ```
   converter.[sh|bat] -target 3.0.5 -exportpath [“./export files”:. |
   “.\export files”;.] MyWallet 0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
   ```

3. Specify the configuration file in the command line when you run the Converter.

   The syntax to specify a configuration file is:

   `converter.[sh | bat] -config` *configuration file name*

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

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

Using a JSON Configuration File for Converter in the Extended Mode

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

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

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

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

Table 4-3 JSON File Options for Converter

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

Table 4-3 (Cont.) JSON File Options for Converter

Option	Level	Description
`integer`	CAP file	Instructs the Converter to support the 32-bit integer type.
`exportPath`	CAP file	Specifies the root directories in which the Converter looks for the export files. The same rules apply for the compact mode. Note that the Java Card API framework export files directory is not required.
`target`	CAP file	Specifies the Java Card platform version on which the CAP file that is generated by the Converter (in the extended mode) is loaded. If the target is not specified, the default value would be the latest release version, that is 3.2.0. The other valid value for the current release is 3.1.0. Extended CAP files are 2.3 version CAP files no matter the target platform used.
`inputPackages`	CAP file	An array of JSON objects, each representing the configuration for the Java package to be converted and added to the CAP file.
`staticResources`	CAP file	An array of JSON objects, each representing the configuration for a static resource to be loaded on the CAP file.
`sign`	CAP file	A JSON object representing the configuration for signing the CAP file, which is generated by the Converter.
`PackageAID`	Package	The AID of the package as defined in the compact mode.
`PackageName`	Package	The fully-qualified name of the package as defined in the compact mode.
`baseDir`	Package	Sets the root directory from where the Converter looks for the classes in the package. If this option is not specified, the Converter uses the location of the configuration file as the root directory.
`outputDir`	Package	Sets the root directory for output of the JCA and EXP files generated for this package. The same rules apply for the compact mode. If this option is not set, the `baseDir` value is taken.
`public`	Package	Specifies if a package is exported or not. The values and its description follow: • If the value is set to `true`, and the package is a library, then all the public classes and interfaces are exported. • If the value is set to `true`, and the package is an applet package, then only shareable interfaces are exported. • If the value is set to `false`, nothing from the package is exported. Also, the AID field of the package will not appear in the header component of the CAP file and the AID field is ignored. Because of this, in the generated JCA files, the AID of a private package will have a random value. For a private package, the EXP file is not generated and the value of the EXP file is ignored.
`version`	Package	The user-defined version of the package as defined in the compact mode. If the package is private and the `exportmap` field is set to `false`, the `version` field is ignored.
`JCA`	Package	Instructs the Converter to write or not to write the JCA file to the disk for the package.

Table 4-3 (Cont.) JSON File Options for Converter

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

Handling Relative Paths

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

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

For example, the static resources are defined as follows (in Unix/Linux OS replace \\ with /):

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

If the JSON configuration file is in the following location (in Unix/Linux OS replace \ with / and a proper Unix/Linux OS path):

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

Then the Converter finds the data for the static resources in the following locations (in Unix/Linux OS replace \ with / and a proper Unix/Linux OS path):

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

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

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

Converter JSON Configuration File Sample

The following is an example of the converter JSON configuration.

Note that if you are using Unix/Linux OS, replace all `\\` references with `/` and all semicolons (`;`) with colons (`:`):

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

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

Validating a JSON Configuration File

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

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

```
converter.[sh|bat] -help JSON > converter_schema.json
```

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

File Naming for the Converter

This section describes the names of input and output files for the Converter, and gives the correct location for these files. With some exceptions, the Converter follows the Java programming language naming conventions for default directories for input and output files. These naming conventions comply with the definitions in the Java Card Virtual Machine specification.

This section includes the following:

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

Input File Naming Conventions

The input files for the Converter are Java class files named with the `.class` suffix. Generally, there are several class files making up a package. All the class files for a package must be located in the same directory under the root directory, following the Java programming language naming conventions. In the compact mode, the root directory can be set from the command line using the `-classdir` option. If this option is not specified, the root directory defaults to the directory from which the user invoked the Converter. In the extended mode, the root directory can be set from the JSON configuration file using the `baseDir` field. This is set for each package contained in the extended CAP file. If the field is not specified for a specific package, the root directory for that package defaults to the directory in which the JSON configuration file resides.

Suppose, for example, you want to convert the package `java.lang`. If you use the `-classdir` flag to specify the root directory as `/home/mywork` (Unix/Linux OS) or `C:\mywork` (Windows OS), the command line is:

```
converter.[sh|bat] -classdir [/home/mywork|C:\mywork] java.lang
package_AID package_version
```

where *package_AID* is the application ID of the package and *package_version* is the user-defined version of the package.

If you use the `baseDir` field to specify the root directory as `/home/mywork` or `C:\mywork`, the JSON field looks like this: `"baseDir":"/home/mywork` for Unix/Linux OS or `"baseDir":"C:\\mywork"` for Windows OS

The Converter looks for all class files in the `java.lang` package in the directory `/home/mywork/java/lang` on Unix/Linux OS or `C:\mywork\java\lang` on Windows OS.

Output File Naming Conventions

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

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

In the above `-classdir` example, by default, the output files are written to the directory

```
[ /home/mywork/java/lang/javacard | C:\mywork\java\lang\javacard ]
```

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

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

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

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

Verification of Input and Output Files

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

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

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

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

Creating a debug.msk Output File

To create a `debug.msk` output file:

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

Using Export Files

A Java Card technology-based export file contains the public API linking information of classes in an entire package. The Unicode string names of classes, methods and fields are assigned unique numeric tokens.

Export files are not used directly on a device that implements a Java Card virtual machine. However, the information in an export file is critical to the operation of the virtual machine on a device. An export file is produced by the Converter when a package is converted. You can use this package's export file later to convert another package that imports classes from the first package. Information in the export file is included in the CAP file of the second package, then is used on the device to link the contents of the second package to items imported from the first package.

During the conversion, when the code in the currently converted package references a different package, the Converter loads the export file of the different package. The Converter also tries to load the shareable interface class files being imported from that package.

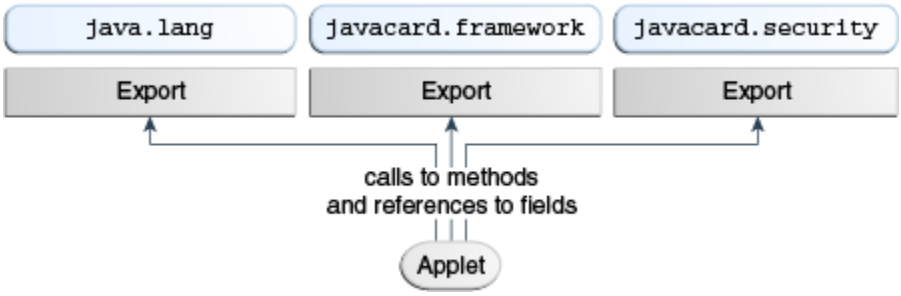
For more information on export files, see Verifying CAP and Export Files .

Calls Between Packages Go Through the Export Files illustrates how an applet package is linked with the `java.lang`, the `javacard.framework` and `javacard.security` packages through their export files.

You can use the `-exportpath` command option and the `exportPath` JSON field to specify the locations of export files and the shareable interface class files. The path consists of a list of root directories in which the Converter looks for export files and shareable interface class files. Export files must be named as the last portion of the package name followed by the extension `.exp`. Export files are located in a subdirectory called `javacard`, following the relative directory path that matches the package name. The shareable interface class files are located in the relative directory path that matches the package name.

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

Figure 4-1 Calls Between Packages Go Through the Export Files



Specifying an Export Map

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

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

  During package reimplementation, the API of the package (exportable classes, interfaces, fields and methods) must remain the same.

- When the minor version increases (package upgrading).

  During a package upgrade, changes that do not break binary compatibility with preexisting packages are allowed (see "Binary Compatibility" in the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.2*).

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

To specify an export map:

1. Set the `-exportmap` command option described in Converter Command Line Arguments when you run the Converter in the compact mode.

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

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

Viewing an Export File as Text

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

To view an export file as text:

- Enter the following command (see exp2text Command Line Options for a description of the options):

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

Table 4-4 exp2text Command Line Options

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

5

Working With CAP Files

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

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

This chapter contains the following sections:

Compact CAP File and Manifest File Syntax

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

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

The information in the manifest file is presented in name:value pairs. These name:value pairs are described in Name:Value Pairs in the MANIFEST.MF File .

Table 5-1 Name:Value Pairs in the MANIFEST.MF File

Name	Value
`Java-Card-CAP-Creation-Time`	Creation time of CAP file. For example: `Tue Jan 15 11:07:55 PST 2006` The format of the time stamp is operating system-dependent.
`Java-Card-Converter-Version`	The version of the converter tool. Default: `3.2.0`.
`Java-Card-Converter-Provider`	Provider of the converter tool. For example: `Oracle Corporation`
`Java-Card-CAP-File-Version`	CAP file *major.minor* version. Possible values are: `2.1`, `2.2`, or `2.3`.
`Java-Card-Package-Version`	The *major.minor* version of package. For example: `1.0`

Table 5-1 (Cont.) Name:Value Pairs in the MANIFEST.MF File

Name	Value
`Java-Card-Package-AID`	AID for the package. For example: `0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07`
`Java-Card-Package-Name`	The fully-qualified package name in dot (.) format. For example: `javacard.framework`
`Java-Card-Applet-<n>-AID`	The AID for applet *n*. For example: `0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x05`
`Java-Card-Applet-<n>-Name`	Simple class name for applet *n*. For example: `MyApplet`
`Java-Card-Import-Package-<n>-AID`	The AID for imported package *n*. For example: `0xa0:0x00:0x00:0x00:0x62:0x00:0x01`
`Java-Card-Import-Package-<n>-Version`	The *major.minor* version of imported package *n*. For example: `1.0`
`Java-Card-Integer-Support-Required`	Can be `TRUE` or `FALSE`. The value is `TRUE` if the package requires integer support.

The properties in the manifest file include:

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

Sample Manifest File

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

```
Manifest-Version: 1.0
Created-By: 17.0.9 (Oracle Corporation)
[...]
Java-Card-CAP-Creation-Time: Tue Feb 13 09:56:03 CET 2024
Java-Card-Converter-Version:  [v3.2.0]
Java-Card-Converter-Provider: Oracle Corporation
Java-Card-CAP-File-Version: 2.3
Java-Card-Package-Version: 1.0
Java-Card-Package-Name: jcard.applications
Java-Card-Package-AID: 0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07
Java-Card-Applet-1-Name: MyClass1
Java-Card-Applet-1-AID:
0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x05
```

```
Java-Card-Applet-2-Name: MyClass2
Java-Card-Applet-2-AID: 0xa0:0x00:0x00:0x00:0x62:0x03:0x01:0x0c:0x07:0x06
Java-Card-Imported-Package-1-AID: 0xa0:0x00:0x00:0x00:0x62:0x00:0x01
Java-Card-Imported-Package-1-Version: 1.0
Java-Card-Imported-Package-2-AID: 0xa0:0x00:0x00:0x00:0x62:0x01:0x01
Java-Card-Imported-Package-2-Version: 1.1
Java-Card-Integer-Support-Required: TRUE
```

Extended CAP File Manifest File Syntax

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

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

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

Table 5-2 Extended CAP File Manifest File Name Syntax

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

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

Table 5-3 Name:Value Pairs in the extended CAP MANIFEST.MF File

Name	Value
`Java-Card-CAP-Name`	The extended CAP file name as defined in the `CAP_name` field from the JSON input configuration file.

Table 5-3 (Cont.) Name:Value Pairs in the extended CAP MANIFEST.MF File

Name	Value
Java-Card-CAP-AID	The extended CAP file AID as present in the header component of the CAP file.
Java-Card-CAP-Version	The extended CAP file version as present in the header component of the CAP file.

Sample Extended CAP Manifest File

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

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

Generating CAP Files From Java Card Assembly Files

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

Running capgen

To run `capgen`:

- Enter the following on the command line (see capgen Command Line Options for a description of the options):

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

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

Table 5-4 capgen Command Line Options

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

Using a JSON Configuration File for capgen in the Extended Mode

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

For using the `capgen` tool in the extended mode, a JSON configuration file must be used with the `-config` option. This JSON file is similar to the one used by the Converter tool (see Using a JSON Configuration File for Converter in the Extended Mode). The only difference is, some of the general conversion parameters that are used by the Converter tool, including the

export paths, are not used by the `capgen` tool. This is because, this information is already present in the JCA files. For each of the package, only the path to the JCA files is provided. The information that is not present in the JCA files remain in the JSON file, similar to the extended CAP file information and static resources information.

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

Table 5-5 JSON File Options for capgen

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

Capgen JSON Configuration File Sample

The `capgen` JSON configuration file sample follows (for Unix/Linux OS replace \\ with /):

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

Producing a Text Representation of a CAP File

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

Running capdump

To run `capdump`:

- Enter the following on the command line:

  `capdump.[sh|bat]` *`filename`*

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

> **Note:**
>
> The file to invoke `capdump` is a shell script/batch file (`capdump.[sh|bat]`) that must be run from a working directory of `[ $JC_HOME_TOOLS/bin | %JC_HOME_TOOLS%\bin ]` in order for the code to execute properly.

6
Verifying CAP and Export Files

This chapter describes off-card verification as a means for evaluating CAP and export files in a desktop environment.

When applied to the set of CAP files that reside on a Java Card technology compliant secure element and the set of export files used to construct those CAP files, the Java Card technology-enabled off-card verifier provides the means to assert that the content of the secure element has been verified.

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

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

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

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

This chapter contains the following sections:

Overview of Verifying CAP and Export Files

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

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

- `verifyrev` - Verifying Binary Compatibility

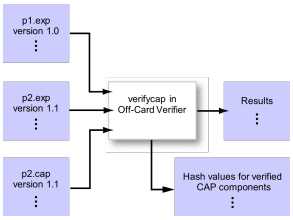
Verifying CAP Files

The `verifycap` tool is used to verify a CAP file within the context of packages' export files (if any) and the export files of imported packages. This verification confirms whether a CAP file is internally consistent, as defined in the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.2*, and consistent with a context in which it can reside in a Java Card technology-enabled device.

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

Each individual export file is verified as a single unit. The scenario is shown in Verifying a CAP File. In the figure, the package `p2` CAP file is being verified. Package `p2` has a dependency on package `p1`, so the export file from package `p1` is also input. The `p2.exp` file is only required if `p2.cap` exports any of its elements.

Figure 6-1 Verifying a CAP File



Running verifycap

The file to invoke `verifycap` is a shell script/batch file (`verifycap.[sh|bat]`) that you must run from a working directory of *JC_HOME_TOOLS*`\bin` in order for the code to execute properly.

To run `verifycap`:

- Enter the following command (verifycap Command Line Arguments describes the available options):

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

Table 6-1 verifycap Command Line Arguments

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

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

Verifying Export Files

The `verifyexp` tool is used to verify an export file as a single unit. This verification is "shallow", examining only the content of a single export file, not including export files of packages referenced by the package of the export file. The verification determines whether an export file is internally consistent and viable as defined in the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.2*.

Figure 6-2 Verifying an Export File

Running verifyexp

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

To run `verifyexp`:

- Enter the following command (verifyexp Command Line Argument describes the available options):

  `verifyexp.[sh|bat] [options] export-file`

Table 6-2 verifyexp Command Line Argument

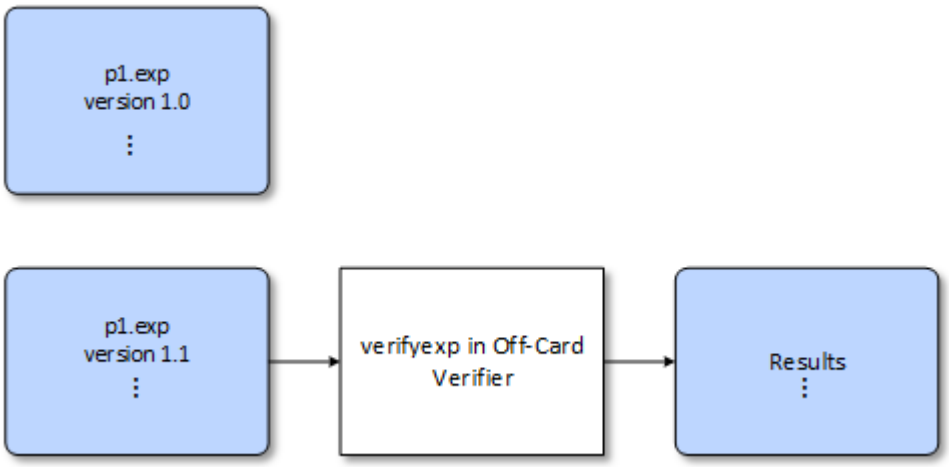
Argument	Description
<export file>	Fully qualified path and name of the export file.

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

Verifying Binary Compatibility

The `verifyrev` tool checks for binary compatibility between revisions of a package by comparing the respective export files. This scenario is illustrated in Verifying Binary Compatibility of Export Files. The export files from version 1.0 and 1.1 of package `p1` are input to `verifyrev`. The verification examines whether the Java Card platform version rules, including those imposed for binary compatibility as defined in the *Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.2*, have been followed.

Figure 6-3 Verifying Binary Compatibility of Export Files



Running verifyrev

The file to invoke `verifyrev` is a shell scrpt/batch file (`verifyrev.[sh|bat]`) that must be run from a working directory of `[ $JC_HOME_TOOLS/bin | %JC_HOME_TOOLS%\bin]` in order for the code to execute properly.

To run `verifyrev`:

- Enter the following command:

  ```
  verifyrev.[sh|bat] [options] export-file export-file
  ```

  The first *export-file* argument on the command line represents the fully qualified path of the export files to be compared, while the second export file name must be the same as the first one with a different path, for example (for Unix/Linux OS replace \ with /):

  ```
  verifyrev.bat d:\testing\old\crypto.exp d:\testing\new\crypto.exp
  ```

Command Line Options for Off-Card Verifier Tools describes additional command-line options for the off-card verifier tools.

Command Line Options for Off-Card Verifier Tools

The `verifycap`, `verifyexp`, and `verifyrev`, off-card verifier tools share many of the same command line options. The only exceptions are the `-package`, `-outfile`, `-digest`, and `-target` options that are available for `verifycap` only.

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

Table 6-3 `verifycap`, `verifyexp`, `verifyrev` Command Line Options

Option	Description
`-help`	Prints help message.
`-nobanner`	Suppresses banner message.
`-nowarn`	Suppresses warning messages.
`-package <package name>`	(*Available for* `verifycap` *only*) Sets the name of the package to be verified.
`-outfile`	(*Available for* `verifycap` *only*) Specifies the name of the output file to store the digest (default: no output file created).
`-digest`	(*Available for* `verifycap` *only*) Specifies the digest to use (default: SHA-256)
`-target`	(*Available for* `verifycap` *only*) Specifies the target platform (3.0.4, 3.0.5, 3.1.0 or 3.2.0). When a target is specified, the `verifier` automatically uses an internal copy of the export files corresponding to the specified version and ignores the export files for platform packages provided on the command line. This ensures that the correct version of export files is used and allows the `verifier` to detect some binary incompatibility issues when extending some of the platform classes or interfaces on versions 3.0.4 and 3.0.5 of the platform. Note that using this option to specify the target still requires that you provide the export files for all other packages used by the CAP file. If no target is specified, the export files for all the packages used by the CAP file must be provided on the command line.
`-verbose`	Enables verbose mode.
`-version`	Prints version number and exit.

Table 6-3 (Cont.) `verifycap`, `verifyexp`, `verifyrev` Command Line Options

Option	Description
`-C` *command-options-file* or `-commandoptionsfile` *command-options-file*	Optional. Specifies a file containing command-line options.

7

Programming for the Large Address Space

This chapter describes two ways in which you can take advantage of large memory storage insmart cards: by using library packages properly and by separating your data properly.

While the extended CAP files allow multiple packages in a single CAP file and method component of size greater than 64 K, the compact CAP files are still limited to a single packageand have a 64 K limit on the method component. Therefore, you must take special considerations when using the compact CAP files to take advantage of the large memory storage in smart cards.

This chapter contains the following sections:

Overview of Programming for the Large Address Space

The default address space automatically built in the simulator is the large address space. Allowing your applications to take advantage of the large address capabilities of the Classic Edition simulator using compact CAP files requires careful planning and programming. Some size limitations still exist within the simulator. The way that you structure large applications and applications that manage large amounts of data determines how the large address space can be exploited.

Programming Large Applications and Libraries

With the introduction of Extended CAP files (see Programming Large Java Card Applications with Multiple Packages) since Java Card, Version 3.1, Java Card facilitates the development of large applications for its platform.

However, there might be scenarios where developers want to continue using the Compact CAP file format. For example, to target Java Card products that do not support Extended CAP files. When using the compact CAP file format, the most important limitation on a package is the 64 KB limitation on the maximum component size. This is especially true for the Method component. If the size of an application's Method component exceeds 64 KB, then the Java Card Converter doesn't process the package and returns an error.

You can overcome the component size limitation by dividing the application into separate application and library components. The Java Card platform has the ability to support library packages. Library packages contain code, which can be linked and reused by several applications. By dividing the functionality of a given application into application and library packages, you can increase the size of the components. It is important to note that there are important differences between library packages and applet packages:

- In a library package, all public fields are available to other packages for linking.

- In an applet package, only interactions through a shareable interface are allowed by the firewall.

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

Handling a Package as a Separate Code Space

Several applications and API functionality can be installed in the smart card simultaneously by handling each package as a separate code space. This technique lets you exceed the 64KB limit, and provide full Java Card API functionality and support for complex applications requiring larger amounts of code.

Storing Large Amounts of Data

Today's applications are required to securely store ever-growing amounts of information about the cardholder or network identity. This information includes certificates, images, security keys, and biometric and biographical information.

This information sometimes requires large amounts of storage. Before version 2.2.2, versions of the Java Card Platform Simulator had to save downloaded applications or user data in valuable persistent memory space. Sometimes, the amount of memory space required was insufficient for some applications. However, the memory access schemes introduced with version 2.2.2 allow applications to store large amounts of information, while still conforming to the Java Card specification.

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

The Java Card Platform Simulator, enables you to use all of the available persistent memory space to store object information. By allowing you to separate data storage into distinct array and object types, this simulator enables you to store the large amounts of data demanded by today's applications.

8

Programming Large Java Card Applications with Multiple Packages

SinceJava Card Version 3.1, multiple Java packages can be bundled into one Java Card CAP file using an extended CAP file format.

This feature enables you to:

- Keep a modular design by having applications or libraries made of multiple packages.
- Distribute an application with the libraries it relies on.
- Control the visibility of each of the packages deployed in a CAP file.
- Overcome the size limitation of 64 K in compact CAP files.

The converter uses the extended CAP file format when multiple packages are used in a single CAP file or when the converted code for the whole CAP file exceeds the size of 64 KB.

The Java Card products optionally supports the extended CAP file. Before using this extended CAP file format, it is important to check if the target Java Card product supports this feature or not.

Refer to *JavaCard Platform Virtual Machine Specification, Classic Edition, Version 3.2*, for more information on the CAP file and its format.

Topics:

CAP File Identification

Each CAP file has an AID and a version. The AID and version values are dependent on the format of the CAP file (compact and extended) as follows:

- When the CAP file contains a unique package and it is in the compact format, its AID and version are same as the AID and version of the package it contains.
- When the CAP file contains multiple packages and it is in the extended format, it has its own AID and version, independent of the AIDs and versions of the packages it contains.

Package Visibility

The extended CAP file format offers more flexibility for the package visibility as follows:

- Each public package inside a CAP file has an AID and version. This AID and version uniquely identify this package when a package in another CAP file is importing it.
- Private packages inside a CAP file or packages that have no exported information (like an applet package with no `Shareable` interfaces) have no AID and version. In addition, because nothing is exported, the Converter does not generate an export file. Packages in other CAP files imports nothing from such packages.
- Unlike in an applet compact CAP file, if an extended CAP file is an applet CAP file, then the public library packages that are contained in the CAP file are exported as if they were present in a compact CAP file. In addition, the public applet packages contained in an extended CAP file are exported individually based on the same rules as for the compact CAP files (only public `Shareable` interfaces).
- Packages inside a bundle are visible to each other and the standard Java access rules (`public`, `protected`, `package`, or `private`) apply, irrespective of whether the packages are `public` or `private`.

Firewall Context

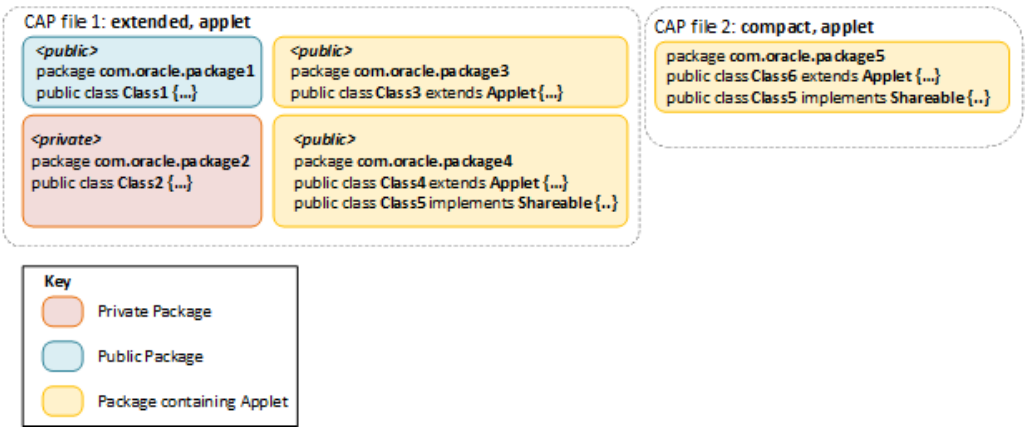
If a CAP file contains at least one package with one or more non-abstract classes that extend the `javacard.framework.Applet` class, then it is associated with a firewall context.

Refer to the Java Card Platform Runtime Environment Specification, Classic Edition, Version 3.2, for more details.

Extended CAP Accessibility Example

To understand the extended CAP file accessibility, let's consider a scenario as shown in the following figure:

Figure 8-1 Extended CAP Accessibiliy Example



The following table describes the package level access under different access conditions (1, 2, and 3):

Table 8-1 Package Level Access

Accessibility	package1	package2	package3	package4	package5
`package1` has access to:	N/A	Yes (1) and (2)	Yes (1)	Yes (1)	Yes (1), (2), and (3)
`package2` has access to:	Yes (1) and (2)	N/A	Yes (1)	Yes (1)	Yes (1), (2), and (3)
`package3` has access to:	Yes (1) and (2)	Yes (1) and (2)	N/A	Yes (1)	Yes (1), (2), and (3)
`package4` has access to:	Yes (1) and (2)	Yes (1) and (2)	Yes (1)	N/A	Yes (1), (2), and (3)
`package5` has access to:	Yes (1) and (2)	No (1)	No (1) and (3)	Yes (1) and (3)	N/A

The following are the access conditions (1, 2, and 3) that are listed in the table:

1. **Exported packages in an extended CAP file** - Packages in an extended CAP file can be marked as `public` or `private`. Only the `public` packages are accessible from packages in another CAP file. However, all packages are accessible within the same CAP file.
2. **Java access rules** - Code access conforms to Java accessibility rules (`private`, `public`, `package`, `protected`, and so on). For example, inside the `Class1` or `Class2` methods, a `Class3()` or a `Class4()` constructor can be called (only if the constructors are `public`) or other public methods from `Class3` and `Class4`, even if `Class3` and `Class4` are Java Card applets.
3. **Java Card access rules for package containing an Applet** - A public package containing a class extending the `javacard.framework.Applet` class does not export all its `public` classes and interfaces. Only the interfaces extending the `javacard.framework.Shareable` interface or the classes implementing it are exported and visible from other packages. For example, code from `package5` can access only the `Class5` in `package4` and cannot access content of `package3` because nothing is exported.

Design Rules for a Java Card Application with Large Method Component

In compact CAP files, the Method Component is limited to a 64 KB size. This can be a constraint if an application has many features, if a library has a large API, or if it is too large to fit into that size after conversion.

The extended CAP file offers a solution to this by creating a Method Component that has a maximum size of eight megabytes. For large applications, the extended mode is preferable.

The Converter splits the large Method Component into blocks, each with a maximum size of 64 KB. It is important to note that methods cannot be divided between two blocks and all exception handlers for a method must be contained in the same block of the method code. Because of this, when programming large Java Card applications for extended CAPs, the method code size must not be too large and specialization pattern must be used whenever possible.

Part II
Appendices

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

- Java Card Assembly Syntax Example
- Additional Optional Ant Tasks

A

Java Card Assembly Syntax Example

This appendix contains two examples of annotated Java Card platform assembly (Java Card Assembly) files that are generated with the Converter. The first example contains the output of 3.0.5 Converter.

The second example highlights the changes in Java Card Assembly (JCA) files generated with the latest Converter. A notable change is that the `publicMethodTable` format has changed due to the Virtual Method Tokens feature (see Section 6.9.2.7 in Java Card Platform Virtual Machine Specification, Classic Edition, Version 3.2, for more details).

The comments in these files are intended to help you understand the syntax of the Java Card Assembly language, and to act as a guide for debugging the Converter output.

> **Note:**
>
> If you are using a source release, you can get an HTML file with the BNF grammar for the Java Card Assembly syntax by using the Java `jjdoc` tool with:
>
> *JC_HOME_TOOLS*`\src\tools\converter\com\sun\javacard\jcasm\Parser.jj`

```
/*
 * Java Card Assembly annotated example. The code
 * contained within this example is not an executable
 * program. The intention of this program is to illustrate the
 * syntax and use of the Java Card Assembly directives and commands.
 *
 * A Java Card Assembly file is textual representation of the
 * contents of a CAP file.
 * The contents of a Java Card Assembly file are hierarchically
 * structured. The format of this structure is:
 *
 *     package
 *         package directives
 *         imports block
 *         applet declarations
 *         constant pool
 *         class
 *             field declarations
 *             virtual method tables
 *             interface table
 *             [remote interface table] - only for remote classes
 *             methods
 *                 method directives
 *                 method statements
 *
 * Java Card Assembly files support both the Java single line
 * comments and Java block
 * comments. Anything contained within a comment is ignored.
 *
```

```
 * Numbers may be specified using the standard Java notation.
 * Numbers prefixed
 * with a 0x are interpreted as
 * base-16, numbers prefixed with a 0 are base-8, otherwise
 * numbers are interpreted
 * as base-10.
 *
*/

/*
 * A package is declared with the .package directive. Only one
 * package is allowed
 * inside a Java Card Assembly
 * file. All directives (.package, .class, et.al) are case
 * insensitive. Package,
 * class, field and
 * method names are case sensitive. For example, the .package
 * directive may be written
 * as .PACKAGE,
 * however the package names example and ExAmPle are different.
 */
.package example {
    /*
     * There are only two package directives. The .aid and .version
     * directives declare
     * the aid and version that appear in the Header Component of
     * the CAP file.
     * These directives are required.
    .aid 0:1:2:3:4:5:6:7:8:9:0xa:0xb:0xc:0xd:0xe:0xf;
        // the AIDs length must be
        // between 5 and 16 bytes inclusive
    .version 0.1;        // major version <DOT> minor version
    /*
     * The imports block declares all of packages that this
     * package imports. The data
     * that is declared
     * in this section appears in the Import Component of the
     * CAP file. The ordering
     * of the entries
     * within this block define the package tokens which must be
     * used within this
     * package. The imports
     * block is optional, but all packages except for java/lang
     * import at least
     * java/lang. There should
     * be only one imports block within a package.
     */
    .imports {
        0xa0:0x00:0x00:0x00:0x62:0x00:0x01 1.0;
        // java/lang aid <SPACE>
        // java/lang major version <DOT> java/lang minor version
        0:1:2:3:4:5 0.1;                    // package test2
        1:1:2:3:4:5 0.1;                    // package test3
        2:1:2:3:4:5 0.1;                    // package test4
    }
    /*
     * The applet block declares all of the applets within
     * this package. The data
     * declared within this block appears
     * in the Applet Component of the CAP file. This section may
     * be omitted if this
```

```
 * package declares no applets. There
 * should be only one applet block within a package.
 */
.applet {
    6:4:3:2:1:0 test1;   // the class name of a class within this
                         // package which
    7:4:3:2:1:0 test2;   // contains the method install([BSB)V
    8:4:3:2:1:0 test3;
}
/*
 * The constant pool block declares all of the constant
 * pool's entries in the
 * Constant Pool Component. The positional
 * ordering of the entries within the constant pool block
 * define the constant pool
 * indices used within this package.
 * There should be only one constant pool block within a package.
 *
 * There are six types of constant pool entries. Each of these
 * entries directly
 * corresponds to the constant pool
 * entries as defined in the Constant Pool Component.
 *
 * The commented numbers which follow each line are the constant
 * pool indexes
 * which will be used within this package.
 */
.constantPool {
    /*
     * The first six entries declare constant pool entries that
     * are contained in
     * other packages.
     * Note that superMethodRef are always declared internal
     * entry.
     */
    classRef    0.0;        // 0  package token 0, class token 0
    instanceFieldRef 1.0.2; // 1  package token 1, class token 0,
                            //    instance field token 2
    virtualMethodRef 2.0.2; // 2  package token 2, class token 0,
                            //    instance field token 2
    classRef    0.3;        // 3  package token 0, class token 3
    staticFieldRef  1.0.4;  // 4  package token 1, class token 0,
                            //      field token 4
    staticMethodRef  2.0.5; // 5  package token 2, class token 0,
                            //      method token 5
    /*
     * The next five entries declare constant pool entries
     * relative to this class.
     *
    classRef    test0;                                      // 6
    instanceFieldRef    test1/field1;                       // 7
    virtualMethodRef    test1/method1()V;                   // 8
    superMethodRef  test9/equals(Ljava/lang/Object;)Z;      // 9
    staticFieldRef    test1/field0;                         // 10
    staticMethodRef    test1/method3()V;                    // 11
}
/*
 * The class directive declares a class within the Class Component
 * of a CAP file.
 * All classes except java/lang/Object should extend an internal
 * or external
```

```
 * class. There can be
 * zero or more class entries defined within a package.
 *
 * for classes which extend a external class, the grammar is:
 * .class modifiers* class_name class_token extends
 * packageToken.ClassToken
 *
 * for classes which extend a class within this package,
 * the grammar is:
 * .class modifiers* class_name class_token extends className
 *
 * The modifiers which are allowed are defined by the Java Card
 * language subset.
 * The class token is required for public and protected classes,
 * and should not be
 * present for other classes.
 */
.class final public test1 0 extends 0.0 {
    /*
     * The fields directive declares the fields within this class.
     * There should
     * be only one fields
     * block per class.
     */
    .fields {
        public static int field0 0;
        public int field1 0;
    }
    /*
     * The public method table declares the virtual methods within
     * this classes
     * public virtual method
     * table. The number following the directive is the method
     * table base (See the
     * Class Component specification).
     *
     * Method names declared in this table are relative to
     * this class. This
     * directive is required even if there
     * are not virtual methods in this class. This is necessary
     * to establish the
     * method table base.
     */
    .publicmethodtable 1 {
        equals(Ljava/lang/Object;)Z;
        method1()V;
        method2()V;
    }
    /*
     * The package method table declares the virtual methods
     * within this classes
     * package virtual method
     * table. The format of this table is identical to the public
     * method table.
     */
    .packagemethodtable 0 {}
    .method public method1()V 1 { return; }
    .method public method2()V 2 { return; }
    .method protected static native method3()V 0 { }
    .method public static install([BSB)V 1 { return; }
}
```

```
.class final public test9 9 extends test1 {
    .publicmethodtable 0 {
        equals(Ljava/lang/Object;)Z;
        method1()V;
        method2()V;
    }
    .packagemethodtable 0 {}
    .method public equals(Ljava/lang/Object;)Z 0 {
        invokespecial 9;
        return;
    }
}
.class final public test0 1 extends 0.0 {
    .Fields {
        // access_flag, type, name [token [static Initializer]] ;
        public static byte field0 4 = 10;
        public static byte[] field1 0;
        public static boolean field2 1;
        public short field4 2;
        public int field3 0;
    }
    .PublicMethodTable 1 {
        equals(Ljava/lang/Object;)Z;
        abc()V;                  // method must be in this class
        def()V;
        labelTest()V;
        instructions()V;
    }
    .PackageMethodTable 0 {
        ghi()V;                  // method must be in this class
        jkl()V;
    }
    // if the class implements more than one interface, multiple
    // interfaceInfoTables will be present.
.implementedInterfaceInfoTable
.interface 1.0 {   // java/rmi/Remote
}
.interface RemoteAccount { // The table contains method tokens
10;  // getBalance()S
9;   // debit(S)V
8;   // credit(S)V
11;  // setAccountNumber([B)V
12;  // getAccountNumber()[B
}
}
.implementedRemoteInterfaceInfoTable { // The table contains
                                       // method tokens
// excluding java.rmi.Remote
.interface RemoteAccount { // Contains method tokens
getBalance()S   10;        // getBalance()S
debit(S)V       9;         // debit(S)V
credit(S)V      8;         // credit(S)V
setAccountNumber([B)V   11;  // setAccountNumber([B)V
getAccountNumber()[B    12;  // getAccountNumber()[B
}
}
    /*
     * Declaration of 2 public visible virtual methods and two
     * package visible
     * virtual methods..
     */
```

```
        .method public abc()V 1 {
            return;
        }
        .method public def()V 2 {
            return;
        }
        .method ghi()V 0x80 {               // per the CAP file
                              //specification, method tokens
                           // for package visible methods
            return; // must have the most significant bit set to 1.
        }
        .method jkl()V 0x81 {
            return;
        }
        /*
         * This method illustrates local labels and exception table
         * entries. Labels
         * are local to each
         * method. No restrictions are placed on label names except
         * that they must
         * begin with an alphabetic
         * character. Label names are case insensitive.
         *
         * Two method directives are supported, .stack and .locals.
         * These
         * directives are used to
         * create the method header for each method. If a method
         * directive is omitted,
         * the value 0 will be used.
         *
         */
        .method public static install([BSB)V 0 {
            .stack 0;
            .locals 0;
l0:
l1:
l2:
l3:
l4:
l5:
            return;
            /*
             * Each method may optionally declare an
             * exception table. The start offset,
             * end offset and handler offset
             * may be specified numerically, or with a
             * label. The format of this table
             * is different from the exception
             * tables contained within a CAP file. In a
             * CAP file, there is no end
             * offset, instead the length from the
             * starting offset is specified. In the Java Card Assembly
             * file an end offset is specified
             * to allow editing of the
             * instruction stream without having to recalculate
             * the exception table
             * lengths manually.
             */
            .exceptionTable {
                // start_offset end_offset handler_offset
                // catch_type_index;
```

```
                10 l4 l5 3;
                l1 l3 l5 3;
            }
        }
        /*
         * Labels can be used to specify the target of a
         * branch as well.
         * Here, forward and backward branches are
         * illustrated.
         */
        .method public labelTest()V 3 {
L1:             goto L2;


L2:             goto L1;


            goto_w L1;


            goto_w L3;



L3:             return;
        }
        /*
         * This method illustrates the use of each Java Card platform
         * instruction for version 3.0.5.
         * Mnemonics are case insensitive.
         *
         * See the Java Card virtual machine specification for
         * the specification of
         * each instruction.
         */
        .method public instructions()V 4 {
            aaload;
            aastore;
            aconst_null;
   aload 0;
   aload_0;
   aload_1;
   aload_2;
   aload_3;
   anewarray 0;
   areturn;
   arraylength;
   astore 0;
   astore_0;
   astore_1;
   astore_2;
   astore_3;
   athrow;
   baload;
   bastore;
   bipush 0;
   bspush 0;
   checkcast 10 0;
   checkcast 11 0;
   checkcast 12 0;
   checkcast 13 0;
```

```
checkcast 14 0;
dup2;
dup;
dup_x 0x11;
getfield_a 1;
getfield_a_this 1;
getfield_a_w 1;
getfield_b 1;
getfield_b_this 1;
getfield_b_w 1;
getfield_i 1;
getfield_i_this 1;
getfield_i_w 1;
getfield_s 1;
getfield_s_this 1;
getfield_s_w 1;
getstatic_a 4;
getstatic_b 4;
getstatic_i 4;
getstatic_s 4;
goto 0;
goto_w 0;
i2b;
i2s;
iadd;
iaload;
iand;
iastore;
icmp;
iconst_0;
iconst_1;
iconst_2;
iconst_3;
iconst_4;
iconst_5;
iconst_m1;
idiv;
if_acmpeq 0;
if_acmpeq_w 0;
if_acmpne 0;
if_acmpne_w 0;
if_scmpeq 0;
if_scmpeq_w 0;
if_scmpge 0;
if_scmpge_w 0;
if_scmpgt 0;
if_scmpgt_w 0;
if_scmple 0;
if_scmple_w 0;
if_scmplt 0;
if_scmplt_w 0;
if_scmpne 0;
if_scmpne_w 0;
ifeq 0;
ifeq_w 0;
ifge 0;
ifge_w 0;
ifgt 0;
ifgt_w 0;
ifle 0;
ifle_w 0;
```

```
iflt 0;
iflt_w 0;
ifne 0;
ifne_w 0;
ifnonnull 0;
ifnonnull_w 0;
ifnull 0;
ifnull_w 0;
iinc 0 0;
iinc_w 0 0;
iipush 0;
iload 0;
iload_0;
iload_1;
iload_2;
iload_3;
ilookupswitch 0 1 0 0;
impdep1;
impdep2;
imul;
ineg;
instanceof 10 0;
instanceof 11 0;
instanceof 12 0;
instanceof 13 0;
instanceof 14 0;
invokeinterface 0 0 0;
invokespecial 3;   // superMethodRef
invokespecial 5;   // staticMethodRef
invokestatic 5;
invokevirtual 2;
ior;
irem;
ireturn;
ishl;
ishr;
istore 0;
istore_0;
istore_1;
istore_2;
istore_3;
isub;
itableswitch 0 0 1 0 0;
iushr;
ixor;
jsr 0;
new 0;
newarray 10;
newarray 11;
newarray 12;
newarray 13;
newarray boolean[];        // array types may be declared numerically or
newarray byte[];        // symbolically.
newarray short[];
newarray int[];
nop;
pop2;
pop;
putfield_a 1;
putfield_a_this 1;
putfield_a_w 1;
```

```
putfield_b 1;
putfield_b_this 1;
putfield_b_w 1;
putfield_i 1;
putfield_i_this 1;
putfield_i_w 1;
putfield_s 1;
putfield_s_this 1;
putfield_s_w 1;
putstatic_a 4;
putstatic_b 4;
putstatic_i 4;
putstatic_s 4;
ret 0;
return;
s2b;
s2i;
sadd;
saload;
sand;
sastore;
sconst_0;
sconst_1;
sconst_2;
sconst_3;
sconst_4;
sconst_5;
sconst_m1;
sdiv;
sinc 0 0;
sinc_w 0 0;
sipush 0;
sload 0;
sload_0;
sload_1;
sload_2;
sload_3;
slookupswitch 0 1 0 0;
smul;
sneg;
sor;
srem;
sreturn;
sshl;
sshr;
sspush 0;
sstore 0;
sstore_0;
sstore_1;
sstore_2;
sstore_3;
ssub;
stableswitch 0 0 1 0 0;
sushr;
swap_x 0x11;
sxor;
     }
 }
 .class public test2 2 extends 0.0 {
     .publicMethodTable 0 {}
equals(Ljava/lang/Object;)Z;
```

```
        .packageMethodTable 0 {}
        .method public static install([BSB)V 0 {
   .stack 0;
   .locals 0;
}
   return;
        }
    }
    .class public test3 3 extends test2 {
    /*
    * Declaration of static array initialization is done the same way
    * as in Java
    * Only one dimensional arrays are allowed in the
    * Java Card platform
    * Array of zero elements, 1 element, n elements
    */
    .fields {
        public static final int[] array0 0 = {}; //  [I
        public static final byte[] array1 1 = {17}; //  [B
        public static short[] arrayn 2 = {1,2,3,...,n}; //  [S
    }
        .publicMethodTable 0 {}
   equals(Ljava/lang/Object;)Z;
        .packageMethodTable 0 {}
        .method public static install([BSB)V 0 {
   .stack 0;
   .locals 0;
   return;
        }
    }
    .interface public test4 4 extends 0.0 {
    }
}


// converted by version  [v3.1.0]
.package package1 {
    .aid 0x1:0x2:0x3:0x4:0x5:0x1;
    .version 1.1;

    .imports {
        0xA0:0x0:0x0:0x0:0x62:0x0:0x1 1.0;     //java/lang
    }

    .constantPool {
        // 0
        staticMethodRef 0.0.0()V;        // java/lang/Object.<init>()V
    }

    .class public Class1 0 extends 0.0 {        // extends java/lang/Object

        .publicMethodTable 1 3 {   // 3 is
CAP22_inheritable_public_method_token_count, see 3.1 JCVMSpec 6.9.2.7
            equals(Ljava/lang/Object;)Z 0;
            m1()S 255;  //m1 defined in Class1 1.0 and 1.1
            m2()S 255;  //m2 defined in Class1 1.0 and 1.1
            m4()S 255;  //m4 defined in Class1 1.1
        }

        .packageMethodTable 0 {
        }
```

```
        .method public <init>()V 0 {
            .stack 1;
            .locals 0;

                L0:    aload_0;
                    invokespecial 0;        // java/lang/Object.<init>()V
                    return;
        }

        .method public m1()S 1 {
            .stack 1;
            .locals 0;

                L0:    sspush 170;
                    sreturn;
        }

        .method public m2()S 2 {
            .stack 1;
            .locals 0;

                L0:    bspush 55;
                    sreturn;
        }

        .method public m4()S 3 {
            .stack 1;
            .locals 0;

                L0:    sspush 221;
                    sreturn;
        }

    }

}

// converted by version  [v3.1.0]
.package package2 {
    .aid 0x1:0x2:0x3:0x4:0x5:0x2;
    .version 1.1;

    .imports {
        0x1:0x2:0x3:0x4:0x5:0x1 1.1;        //package1
        0xA0:0x0:0x0:0x0:0x0:0x62:0x0:0x1 1.0;        //java/lang
    }

    .constantPool {
        // 0
        staticMethodRef 0.0.0()V;        // package1/Class1.<init>()V
    }

    .class public Class2 0 extends 0.0 {        // extends package1/Class1

        .publicMethodTable 2 4 {   // 4 is
CAP22_inheritable_public_method_token_count, see 3.1 JCVMSpec 6.9.2.7
            equals(Ljava/lang/Object;)Z 0;
            m1()S 1;     //inherited from Class1 1.0
            m2()S 2;     //method overridden in Class2 1.1
            m3()S 255;   //defined in Class2 1.0
            m4()S 3;     //inherited from Class1 1.1
```

```
            m5()S 255;   //defined in Class2 1.1
        }

        .packageMethodTable 0 {
        }

        .method public <init>()V 0 {
            .stack 1;
            .locals 0;

                L0:    aload_0;
                    invokespecial 0;        // package1/Class1.<init>()V
                    return;
        }

        .method public m2()S 2 {
            .stack 1;
            .locals 0;

                L0:    sspush 187;
                    sreturn;
        }

        .method public m3()S 3 {
            .stack 1;
            .locals 0;

                L0:    sspush 204;
                    sreturn;
        }

        .method public m5()S 5 {
            .stack 1;
            .locals 0;

                L0:    sspush 238;
                    sreturn;
        }

    }

}
```

B

Additional Optional Ant Tasks

This appendix contains a description of the optional Ant tasks supported by this development kit.
The command line tools in this development kit execute Apache Ant transparently, so you are not required to use Ant directly to use the command line tools themselves. Those Ant tasks are required to install and run the development kit.

This development kit also includes additional, optional Apache Ant tasks for skilled Ant users to streamline using the development kit. These optional Ant tasks grouping several command line tools into a single Ant task. This chapter describes how to use these additional, optional, and unsupported Apache Ant tasks.

This chapter includes the following sections:

Location and Installation

The optional Ant tasks are included at:

- *JC_HOME_TOOLS*`\lib\jctasks_tools.jar`

> **Note:**
>
> Use of the additional Ant tasks described in this section is strictly optional and is not formally supported, nor has it been fully tested.

Installing the Ant Tasks

1. Be sure Ant is configured as described in Installing the Java Card Development Kit Tools .
2. Copy the file *JC_HOME_TOOLS*`\lib\jctasks_tools.jar` to a directory that serves as your Ant tasks home directory.
3. Add the `jctasks_tools.jar` file to your classpath or put it into the *Ant-Home-Path*`\lib` directory to be automatically be picked up when Ant is run.

   Where:

   a. *Ant-Home-Path* is the path to the Ant installation.

   b. The value of the *ANT_HOME* environment variable is properly configured to run Ant (see Installing the Java Card Development Kit Tools).

Setting Up the Optional Ant Tasks

The following XML must be added to your `build.xml` file to use the optional Ant tasks in your build.

```
<!-- Definitions for tasks for Java Card tools -->
<taskdefname="capdump"
classname="com.sun.javacard.ant.tasks.CapdumpTask" />
<taskdefname="capgen"
classname="com.sun.javacard.ant.tasks.CapgenTask" />
<taskdef name="exp2text"
classname="com.sun.javacard.ant.tasks.Exp2TextTask" />
<taskdefname="convert"
classname="com.sun.javacard.ant.tasks.ConverterTask" />
<taskdef name="verifyexport"
classname="com.sun.javacard.ant.tasks.VerifyExpTask" />
<taskdef name="verifycap"
classname="com.sun.javacard.ant.tasks.VerifyCapTask" />
<taskdef name="verifyrevision"
classname="com.sun.javacard.ant.tasks.VerifyRevTask" />
<typedef name="appletnameaid"
classname="com.sun.javacard.ant.types.AppletNameAID" />
<typedef name="jcainputfile"
classname="com.sun.javacard.ant.types.JCAInputFile" />
<typedef name="exportfiles"
classname="org.apache.tools.ant.types.FileSet" />
```

Library Dependencies

The JAR files located in *JC_HOME_SIMULATOR*`\lib\tools_simulator.jar` and *JC_HOME_TOOLS*`\lib\tools.jar` contain the libraries required to execute the optional ant tasks. These JAR files must be in the classpath during build execution.

Custom Types

This section includes the following information and description about available custom types:

- AppletNameAID
- JCAInputFile
- ExportFiles

AppletNameAID

`AppletNameAID` groups together name and AID for a Java Card applet.

Table B-1 Parameters for AppletNameAID

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

Example

To use these examples:

- Enter the following example code to set the fully qualified name and AID for the `HelloWorld` applet:

```
<AppletNameAID
appletname="com.sun.javacard.samples.HelloWorld.HelloWorld"
aid="0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1:0x1"/>
```

JCAInputFile

This type is a simple wrapper for a fully qualified JCA file name or a name of an input file that contains a list of input JCA files. In case the input file contains a list of input JCA files, the name of the file should be prepended with "@".

Table B-2 Parameters for JCAInputFile

Attribute	Description	Required
`inputfile`	Fully qualified name of the input file	Yes

Examples

To use these examples:

1. Enter the following example code to set the fully qualified name of an input JCA file.

```
<jcainputfile
   inputfile="C:\jcas\common\com\sun\javacard\installer
\javacard\installer.jca" />
```

2. Enter the following example code to set the fully qualified name of an input file that contains a list of JCA files.

```
<jcainputfile inputfile="@C:\jc\jcaDemo.in" />
```

ExportFiles

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

Examples

To use these examples:

1. Enter the following example code to set the fully qualified name of an input EXP file:

```
<exportfiles
   file="C:\samples\classes\com\sun\javacard\samples
\HelloWorld\javacard\HelloWorld.exp" />
```

2. Enter the following example code to group all the files in the directory `${server.src}` that are EXP files and do not have the text `Test` in their names:

```
<exportfiles dir="${server.src}">
  <include name="**/*.exp"/>
  <exclude name="**/*Test*"/>
</exportfiles>
```

Ant Task Descriptions

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

- CapDump
- Capgen
- Converter
- Exp2Text
- VerifyCap
- VerifyExp
- VerifyRev

CapDump

Run the CapDump tool to dump the contents of a CAP file.

Table B-3 Parameters for CapDump

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

Capgen

Runs Capgen to generate a CAP file from a JCA file.

Table B-4 Parameters for Capgen

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

Converter

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

Table B-5 Parameters for Converter

Attribute	Description	Required
PackageName	Fully qualified name of the package being converted.	Yes, if the configuration file is not provided.
PackageAID	AID of the package being converted.	Yes, if the configuration file is not provided.
MajorMinorVersion	Major and Minor version numbers of the package, for example, 1.2 (where 1 is major version number and 2 is minor version number).	Yes, if the configuration file is not provided.
CAP	If enabled, tells the converter to create a CAP file.	No
EXP	If enabled, tells the converter to create a EXP file.	No
JCA	If enabled, tells the converter to create a JCA file.	No
ClassDir	The root directory of the class hierarchy. Specifies the directory where the converter looks for class files.	No
Int	If enabled, turns on support for the 32-bit integer type.	No
Debug	If enabled, enables generation of debugging information.	No
ExportPath	Root directories where the Converter looks for export files.	No

Table B-5 (Cont.) Parameters for Converter

Attribute	Description	Required
ExportMap	If enabled, tells the converter to use the token mapping from the pre-defined export file of the package being converted. The converter looks for the export file in the exportpath.	No
Outputdirectory	Sets the output directory where the output files are placed.	No
Verbose	If enabled, enables verbose converter output.	No
noWarn	If enabled, instructs the Converter to not report warning messages.	No
Mask	If enabled, tells the Converter that this package is for mask, so restrictions on native methods are relaxed.	No
NoVerify	If enabled, tells the Converter to turn off verification. Verification is turned on by default.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to `true` if you do not want the Capgen banner showing.	No
version	Prints Converter version number.	No
ConfigFile	Configuration file containing all the configuration parameters for the converter.	No

In addition to the parameters specified in the preceding table, the target Java Card platform can be specified for the converter through the environment variable *JC_TARGET_PLATFORM*. If this environment variable is set, then the converter creates the CAP files for the specified target platform.

Exp2Text

Run Exp2Text tool to convert the export file of a package to a text file.

Table B-6 Parameters for Exp2Text

Attribute	Description	Required
PackageName	Fully qualified name of the package.	Yes
ClassDir	Root directory where the exp2text tool looks for the export file. If no ClassDir is specified, the directory in which the Java VM is invoked is taken as base dir.	No
OutputDir	The root directory for output.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No

Table B-6 (Cont.) Parameters for Exp2Text

Attribute	Description	Required
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to `true` if you do not want the Exp2Text banner showing.	No
version	Prints Exp2Text version number.	No

VerifyCap

Runs off-card Java Card platform CAP file verifier to verify a CAP file. The Java Card platform off-card verifier is invoked in a separate instance of Java VM.

Table B-7 Parameters for VerifyCap

Attribute	Description	Required
CapFile	Fully qualified path and name of CAP file that is to be verified.	Yes
PkgName	Fully qualified Name of the package inside the CAP file for which the CAP file was generated.	No
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose verifier output.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
outFile	Fully qualified output path of the digest file, which contains the digests generated using the default algorithm (SHA-256) for all CAP file components.	
nobanner	Set this element to `true` if you want to suppress Verifier banner.	No
version	Prints the version number of the off-card verifier.	No

VerifyExp

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

Table B-8 Parameters for VerifyExp

Attribute	Description	Required
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose verifier output.	No
classpath	Classpath to use for this task. If required JAR files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No

Table B-8 (Cont.) Parameters for VerifyExp

Attribute	Description	Required
nobanner	Set this element to `true` if you want to suppress Verifier banner.	No
version	Prints the version number of off-card verifier.	No

VerifyRev

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

Table B-9 Parameters for VerifyRev

Attribute	Description	Required
noWarn	If enabled, tells the verifier not to output any warning messages.	No
Verbose	If enabled, enables verbose verifier output.	No
classpath	Classpath to use for this task. If required jar files are not already in the system classpath, you can specify this attribute to put them in the classpath when this task is executed.	No
dir	The directory to invoke the Java VM in.	No
nobanner	Set this element to `true` if you want to suppress Verifier banner.	No
version	Prints the version number of off-card verifier.	No

Glossary

active applet instance

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

AID (application identifier)

defined by ISO 7816, a string used to uniquely identify card applications and certain types of files in card file systems. An AID consists of two distinct pieces. A 5 byte RID (resource identifier) and a 0 to 11byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies.

A unique AID is assigned to each CAP file and public packages in a CAP file. In addition, a unique AID is assigned to each applet in the CAP file. The AID for the CAP file, the package AID of every public package in a CAP file, and the default AID for each applet defined in the CAP file are specified. They are supplied to the converter when the CAP file is generated.

APDU

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

API

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

applet

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

applet application

an application that consists of one or more applets.

applet framework

an API that enables applet applications to be built.

applet developer

a person creating an applet using Java Card technology.

applet execution context

currently active applet owner identifier.

applet firewall

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

applet CAP file

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

applet package

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

assigned logical channel

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

atomic operation

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

atomicity

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

ATR

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

authentication

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

basic logical channel

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

big-endian

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

binary compatibility

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

bytecode

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

CAD

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

CAP file

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

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

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

CAP file component

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

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

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

cast

the explicit conversion from one data type to another.

card session

a card session begins when it is powered up or reset. The card is then able to exchange messages with external clients. The card session ends when the card loses power or is reset.

client application

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

constant pool

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

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

context

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

context switch

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

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

converter

a piece of software that preprocesses all of the Java programming language class files contained in a set of packages and converts them into a CAP file. The Converter also produces *export files* for exported packages.

currently active context

when an object instance method is invoked, an owning context of the object becomes the currently active context.

currently selected applet

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

custom CAP file component

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

default applet

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

EEPROM

an acronym for Electrically Erasable, Programmable Read Only Memory.

entry point method

well-defined method of an object owned by an application (respectively the Java Card RE) that can be "legally" invoked by another application or the Java Card RE (respectively an application). SIO methods and other container-managed objects' lifecycle methods are application entry point methods. Java Card RE entry point objects' methods are Java Card RE entry point methods.

entry point objects

see Java Card RE entry point object.

export file

a file produced by the Converter tool used during classic applet application development that represents the fields and methods of a package that can be imported by classes in other classic applet applications and classic libraries.

externally visible

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

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

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

finalization

the process by which a Java virtual machine (JVM) allows an unreferenced object instance to release non-memory resources (for example, close and open files) prior to reclaiming the object's memory. Finalization is only performed on an object when that object is ready to be garbage collected (meaning, there are no references to the object).

Finalization is not supported by the Java Card virtual machine. The method `finalize()` is not called automatically by the Java Card virtual machine.

firewall

the mechanism that prevents unauthorized accesses to objects in one application group context from another application group context.

flash memory

a type of persistent mutable memory. It is more efficient in space and power than EPROM. Flash memory can be read bit by bit but can be updated only as a block.

Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole.

framework

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

garbage collection

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

global array

an array objects accessible from any context.

group context

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

heap

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

installer

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

installation program

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

instance variables

also known as non-static fields.

instantiation

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

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

instruction

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

internally visible

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

JAR file

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

Java Card Platform Remote Method Invocation

a subset of the Java Platform Remote Method Invocation (RMI) system optionally supported by the Java Card RE. It provides a mechanism for a client application to invoke a method on a remote object of an applet on the card.

Java Card Runtime Environment (Java Card RE)

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

Java Card Virtual Machine (Java Card VM)

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

Java Card RE context

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

Java Card RE entry point object

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

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

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

JDK software

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

library CAP file

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

library package

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

local variable

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

logical channel

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

MAC

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

mask production (masking)

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

method

a procedure or routine associated with one or more classes in object-oriented languages.

multiselectable applets

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

multiselected applet

an applet instance that is selected and, therefore, active on more than one logical channel in the APDU application environment simultaneously.

namespace

a set of names in which all names are unique.

native method

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

nibble

four bits.

non-volatile memory

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

object-oriented

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

object

in object-oriented programming, unique instance of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

origin logical channel

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

owning context

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

owner context

see owning context.

package

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

PCD

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

persistent object

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

PIX

see AID (application identifier).

RAM (random access memory)

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

reference implementation

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

remote interface

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

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

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

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

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.