Runtime Environment Specification

Java Card™ Platform, Version 2.2.1

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A. 650-960-1300

October, 2003

Java CardTM Specification ("Specification")
Version: 2.2.1
Release: October, 2003

4150 Network Circle, Santa Clara, California 95054, U.S.A.

NOTICE

TRADEMARKS

DISCLAIMER OF WARRANTIES

LIMITATION OF LIABILITY

RESTRICTED RIGHTS LEGEND

REPORT

GENERAL TERMS

Contents

Preface

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

This document is a specification of the Java Card™ platform, version 2.2.1 Runtime Environment ("Java Card Runtime Environment" or "Java Card RE"). A vendor of a Java Card technology-enabled device provides an implementation of the Java Card RE. A Java Card RE implementation within the context of this specification refers to a vendor's implementation of the virtual machine (VM) for the Java Card platform ("Java Card virtual machine" or "Java Card VM"), the Java Card Application Programming Interface (API), or other component, based on the Java Card technology specifications. A "reference implementation" is an implementation produced by Sun Microsystems, Inc. Application software written for the Java Card platform is referred to as a Java Card technology-based applet ("Java Card applet" or "card applet").

Who Should Use This Specification?

This specification is intended to assist implementers of the Java Card RE in creating an implementation, developing a specification to extend the Java Card technology specifications, or in creating an extension to the Runtime Environment for the Java Card platform. This specification is also intended for Java Card applet developers who want a greater understanding of the Java Card technology specifications.

Before You Read This Specification

Before reading this guide, you should be familiar with the Java programming language, the Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology is the Sun Microsystems, Inc. website, located at:

http://java.sun.com

How This Specification Is Organized

Chapter 1 "Introduction," gives an overview of the information contained in this specification.

Chapter 2 "Lifetime of the Java Card Virtual Machine," defines the lifetime of the Java Card virtual machine.

Chapter 3 "Java Card Applet Lifetime," defines the lifetime of an applet.

Chapter 4 "Logical Channels and Applet Selection," describes how the Java Card RE handles applet selection.

Chapter 5 "Transient Objects," describes the properties of transient objects.

Chapter 6 "Applet Isolation and Object Sharing," describes applet isolation and object sharing.

Chapter 7 "Transactions and Atomicity," describes the functionality of atomicity and transactions.

Chapter 8 "Remote Method Invocation Service," describes the server-side (card-side) functionality of the Remote Method Invocation (RMI) feature of Java Card platform, version 2.2.1.

Chapter 9 "API Topics," describes API functionality required of a Java Card RE but not completely specified in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1*.

Chapter 10 "Virtual Machine Topics," describes virtual machine specifics.

Chapter 11 "Applet Installation and Deletion," provides an overview of the Applet Installer and Java Card RE required behavior.

Chapter 12 "API Constants," provides the numeric value of constants that are not specified in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1*.

"Glossary," provides definitions of selected terms used in this specification.

Related Books

References to various documents or products are made in this guide. You should have the following documents available:

- *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *The Java™ Language Specification* by James Gosling, Bill Joy, and Guy L. Steele (Addison-Wesley, 1996).
- *The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999).
- *The Java™ Remote Method Invocation Specification* , Sun Microsystems, Inc. (http://java.sun.com/products/jdk/rmi)
- *The Java Class Libraries: An Annotated Reference, Second Edition (Java Series)* by Patrick Chan, Rosanna Lee and Doug Kramer (Addison-Wesley, 1999).
- *ISO 7816 Specification* Parts 1-6. (http://www.iso.org)
- EMV '96 Integrated Circuit Card Specification for Payment Systems Version 3.0 EMV 2000 Integrated Circuit Card Specification for Payment Systems Version 4.0 (http://www.emvco.com)

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Accessing Sun Documentation Online

The Java Developer Connection℠ web site enables you to access Java™ platform technical documentation on the Web:

http://developer.java.sun.com/developer/infodocs/

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

docs@java.sun.com

CHAPTER 1

Introduction

The Java Card™ platform, version 2.2.1 Runtime Environment contains the Java Card virtual machine (VM), the Java Card Application Programming Interface (API) classes (and industry-specific extensions), and support services.

This document, the *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1*, specifies the Java Card RE functionality required by the Java Card technology. Any implementation of Java Card technology shall provide this necessary behavior and environment.

CHAPTER 2

Lifetime of the Java Card Virtual Machine

In a PC or workstation, the Java virtual machine runs as an operating system process. When the OS process is terminated, the Java-language applications and their objects are automatically destroyed.

In Java Card technology the execution lifetime of the virtual machine (VM) is the lifetime of the card. Most of the information stored on a card shall be preserved even when power is removed from the card. Persistent memory technology (such as EEPROM) enables a smart card to store information when power is removed. Since the VM and the objects created on the card are used to represent application information that is persistent, the Java Card VM appears to run forever. When power is removed, the VM only stops temporarily. When the card is next reset, the VM starts up again and recovers its previous object heap from persistent storage.

Aside from its persistent nature, the Java Card virtual machine is just like the Java virtual machine.

The card initialization time is the time after masking, and prior to the time of card personalization and issuance. At the time of card initialization, the Java Card RE is initialized. The framework objects created by the Java Card RE exist for the lifetime of the virtual machine. Because the execution lifetime of the virtual machine and the Java Card RE framework span CAD sessions of the card, the lifetimes of objects created by applets will also span CAD sessions. (CAD means Card Acceptance Device, or card reader. Card sessions are those periods when the card is inserted in the CAD, powered up, and exchanging streams of APDUs with the CAD. The card session ends when the card is removed from the CAD.) Objects that have this property are called persistent objects.

The Java Card RE implementer shall make an object persistent when:

- The `Applet.register` method is called. The Java Card RE stores a reference to the instance of the applet object. The Java Card RE implementer shall ensure that instances of class `applet` are persistent.
- A reference to an object is stored in a field of any other persistent object or in a class's static field. This requirement stems from the need to preserve the integrity of the Java Card RE's internal data structures.

CHAPTER 3

Java Card Applet Lifetime

For the purposes of this specification, applet refers to an applet written for the Java Card platform. An applet instance's lifetime begins when it is successfully registered with the Java Card RE via the `Applet.register` method. Applets registered with the `Applet.register` method exist until deleted by the Applet Deletion Manager (Section 11.3 "The Applet Deletion Manager.") The Java Card RE initiates interactions with the applet via the applet's public methods `install`, `select`, `deselect`, and `process`. An applet shall implement the static `install(byte[], short, byte)` method. If the `install(byte[], short, byte)` method is not implemented, the applet's objects cannot be created or initialized. A Java Card RE implementation shall call an applet's `install`, `select`, `deselect`, and `process` methods as described below.

When the applet is installed on the smart card, the static `install(byte[], short, byte)` method is called once by the Java Card RE for each applet instance created. The Java Card RE shall not call the applet's constructor directly.

3.1 The Method install

When the `install(byte[], short, byte)` method is called, the applet instance has not yet been created. The main task of the `install` method within the applet is to create an instance of the `Applet` subclass using its constructor, and to register the instance. All other objects that the applet will need during its lifetime can be created as is feasible. Any other preparations necessary for the applet to be selected and accessed by a CAD also can be done as is feasible. The `install` method obtains initialization parameters from the contents of the incoming byte array parameter.

Typically, an applet creates various objects, initializes them with predefined values, sets some internal state variables, and calls either the `Applet.register()` method or the `Applet.register(byte[], short, byte)` method to specify the AID (applet IDentifier as defined in ISO 7816-5) to be used to select it. This installation is considered successful when the call to the `Applet.register` method completes without an exception. The installation is deemed unsuccessful if

the `install` method does not call the `Applet.register` method, or if an exception is thrown from within the `install` method prior to the `Applet.register` method being called, or if the `Applet.register` method throws an exception. If the installation is unsuccessful, the Java Card RE shall perform all cleanup when it regains control. That is, all atomically updated persistent objects shall be returned to the state they had prior to calling the `install` method. If the installation is successful, the Java Card RE can mark the applet as available for selection.

Only one applet instance can be successfully registered each time the Java Card RE calls the `Applet.install` method.

3.2 The Method select

Applets remain in a suspended state until they are explicitly selected. Selection occurs when the Java Card RE receives a SELECT FILE APDU in which the name data matches the AID of the applet. Applet selection can also occur on a MANAGE CHANNEL OPEN command. Selection causes an applet to become the currently selected applet. For more details, see Section 4.5 "Applet Selection".

Prior to calling `select`, the Java Card RE shall deselect the previously selected applet. The Java Card RE indicates this to the applet by invoking the applet's `deselect` method or, if multiply selected on more than one logical channel, its `MultiSelectable.deselect` method (for more details, see Section 4.2 "Multiselectable Applets").

The Java Card RE informs the applet of selection by invoking its `select` method or, if being multiply selected on more than one logical channel, its `MultiSelectable.select` method (for more details, see Section 4.2 "Multiselectable Applets").

The applet may decline to be selected by returning `false` from the call to the `select` method or by throwing an exception. If the applet returns `true`, the actual SELECT FILE APDU command is supplied to the applet in the subsequent call to its `process` method, so that the applet can examine the APDU contents. The applet can process the SELECT FILE APDU command exactly like it processes any other APDU command. It can respond to the SELECT FILE APDU with data (see Section 3.3 "The Method process" for details), or it can flag errors by throwing an `ISOException` with the appropriate returned status word. The status word and optional response data are returned to the CAD.

The `Applet.selectingApplet` method shall return true when called during the `select` method. The `Applet.selectingApplet` method will continue to return true during the subsequent `process` method, which is called to process the SELECT FILE APDU command.

If the applet declines to be selected, the Java Card RE will return an APDU response status word of `ISO7816.SW_APPLET_SELECT_FAILED` to the CAD. Upon selection failure, the Java Card RE state is set to indicate that no applet is selected. (See Section 4.5 “Applet Selection” for more details).

After successful selection, all subsequent APDUs directed to the assigned logical channel are delivered to the currently selected applet via the `process` method.

3.3 The Method process

All APDUs are received by the Java Card RE and pre-processed. All commands, except for the MANAGE CHANNEL command result in an instance of the APDU class containing the command being passed to the `process(APDU)` method of the currently selected applet.

Note – A SELECT FILE APDU might cause a change in the currently selected applet prior to the call to the `process` method. (The actual change occurs before the call to the `select` method).

On normal return, the Java Card RE automatically appends 0x9000 as the completion response status word to any data already sent by the applet.

At any time during process, the applet may throw an `ISOException` with an appropriate status word, in which case the Java Card RE catches the exception and returns the status word to the CAD.

If any other exception is thrown during process, the Java Card RE catches the exception and returns the status word `ISO7816.SW_UNKNOWN` to the CAD.

3.4 The Method deselect

When the Java Card RE receives a SELECT FILE APDU command in which the name matches the AID of an applet, the Java Card RE calls the `deselect` method of the currently selected applet or, if multiply selected on more than one logical channel, its `MultiSelectable.deselect` method (for more details see Section 4.2 “Multiselectable Applets”). Applet deselection may also be requested by the MANAGE CHANNEL CLOSE command (For more details, see Section 4.6 “Applet Deselection”). The `deselect` method allows the applet to perform any cleanup operations that may be required in order to allow some other applet to execute.

The `Applet.selectingApplet` method shall return false when called during the `deselect` method. Exceptions thrown by the `deselect` method are caught by the Java Card RE, but the applet is deselected.

3.5 The Method uninstall

This method is defined in the `javacard.framework.AppletEvent` interface. When the Java Card RE is preparing to delete the applet instance, the Java Card RE calls this method, if implemented by the applet, to inform it of the deletion request. Upon return from this method, the Java Card RE will check for reference dependencies before deleting the applet instance.

This method may be called multiple times, once for each applet deletion attempt.

3.6 Power Loss and Reset

Power loss occurs when the card is withdrawn from the CAD or if there is some other mechanical or electrical failure. When power is reapplied to the card and on card reset (warm or cold) the Java Card RE shall ensure that:

- Transient data is reset to the default value.
- The transaction in progress, if any, when power was lost (or reset occurred) is aborted.
- The applet instance that was selected when power was lost (or reset occurred) becomes implicitly deselected. (In this case the `deselect` method is not called.)
- If the Java Card RE implements default applet selection (see Section 4.1 "The Default Applets"), the default applet is selected as the active applet instance for the basic logical channel (channel 0), and the default applet's `select` method is called. Otherwise, the Java Card RE sets its state to indicate that no applet is active on the basic logical channel.

CHAPTER 4

Logical Channels and Applet Selection

Java Card platform, version 2.2.1, provides support for logical channels: the ability to allow a terminal to open up to four sessions into the smart card, one session per logical channel. (Logical channels functionality is described in detail in ISO 7816-4.)

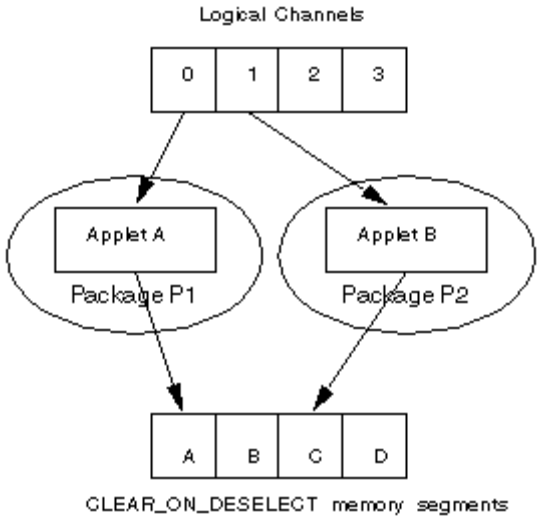
Cards receive requests for service from the CAD in the form of APDUs. The SELECT FILE APDU and MANAGE CHANNEL OPEN APDU are used by the Java Card RE to designate the *active applet instance* for a logical channel session. Once selected, an applet instance receives all subsequent APDUs dispatched to that logical channel, until the applet instance becomes deselected.

A new applet, written for version 2.2.1 of the Java Card platform, can be designed to take advantage of logical channel support. Such an applet can take advantage of multi-session functionality, and can be concurrently selected alongside another applet on a different logical channel and even be selected multiple times simultaneously on different logical channels. As shown in FIGURE 1, an implementation may support from 1 to 4 logical channels, each with its own distinct `CLEAR_ON_DESELECT` memory segment.

Only one logical channel, logical channel 0 (the *basic logical channel*) is active on card reset. A MANAGE CHANNEL APDU command may be issued on this logical channel to instruct the card to open a new logical channel. Applet instances can be selected on different logical channels using the SELECT FILE APDU command, just as they would in a single logical channel environment. The MANAGE CHANNEL APDU command is also used for closing a logical channel. Note that the basic logical channel is permanent and can never be closed.

Legacy applets (written for version 2.1 of the Java Card platform) running on version 2.2.1 need not be aware of logical channel support (and they shall still work correctly). The Java Card RE must guarantee that an applet that was not designed to be aware of multiple sessions is not selected more than once or concurrently with another applet from the same package.

FIGURE 1 Logical Channels for distinct applets



Support for multiple logical channels (with multiple selected applet instances) requires a change to the Java Card platform version 2.1 concept of *selected applet*. Since more than one applet instance can be selected at the same time, and one applet instance can be selected on different logical channels simultaneously, it is necessary to differentiate the state of the applet instances in more detail.

An applet instance will be considered an *active applet instance* if it is currently selected in at least one logical channel, up to a maximum of 4. Each active applet instance from a distinct package executes with a distinct `CLEAR_ON_DESELECT` transient RAM space (see FIGURE 1). An applet instance is the *currently selected applet instance* only if it is processing the current command. There can only be one currently selected applet instance at a given time.

Applets having the capability of being selected on multiple logical channels at the same time, or accepting other applets belonging to the same package being selected simultaneously, are referred to as multiselectable applets. (Refer to FIGURE 2 below.)

No applet is active on the new (or only) logical channel when one of the following occurs:

- The card is reset and no applet has been designated as the default applet instance for the basic channel, or the default applet instance for the basic channel rejects selection.
- A MANAGE CHANNEL OPEN command on the basic channel opens a new channel, and no applet has been designated as the *default applet instance* for that logical channel.

- A new logical channel is opened when a MANAGE CHANNEL OPEN command is issued on a logical channel other than the basic channel, on which there is no active applet.
- A SELECT FILE command fails when attempting to select an applet instance.

4.1 The Default Applets

Normally, applet instances become selected only via a successful SELECT FILE command. However, some smart card CAD applications require that there be a *default card applet instance* to become implicitly selected after every card reset. In addition, some CAD applications may also require a default applet selection when a new logical channel is opened.

The Java Card platform allows the card implementer to designate a *default applet instance* for each of the logical channels supported by the card. For any logical channel, the card implementation may designate an applet instance as the default applet instance for that logical channel. Alternatively, for any logical channel, the implementation may choose to designate no default applet instance at all. Logical channels may share the same applet instance as the default applet instance for more than one channel.

Upon card reset, only the *basic logical channel* (channel 0) is automatically opened. The default card applet instance, if any, is therefore the default applet instance for logical channel 0.

The card reset behavior is:

1. After card reset (or power on, which is a form of reset) the Java Card RE performs its initialization and checks to see if its internal state indicates that a particular applet instance is the default applet instance for the basic logical channel. If so, the Java Card RE makes this applet instance the currently selected applet instance on the basic logical channel, and the applet's `select` method is called. If this method throws an exception or returns `false`, then the Java Card RE sets its state to indicate that no applet is active on the basic logical channel.

   When a default card applet instance becomes active upon card reset, it shall not require its `process` method to be called. The applet instance's `process` method is not called during default applet selection because there is no SELECT FILE APDU.

2. The Java Card RE ensures that the ATR has been sent and the card is now ready to accept APDU commands.

The default applet selection behavior on opening a new channel is:

When a MANAGE CHANNEL command is issued on the basic logical channel and a new logical channel is opened, the Java Card RE checks if there is a designated default applet instance for the newly opened logical channel. If so, the Java Card

RE makes this applet instance the currently selected applet instance on the new logical channel, and the applet's `select` method (`MultiSelectable.select` method if required) is called. If this method throws an exception or returns false, then the Java Card RE closes the new logical channel. (The applet instance's `process` method is not called during default applet selection, because there is no SELECT FILE APDU). A default applet instance shall not require its `process` method to be called.

If a default applet instance was successfully selected, then APDU commands can be sent directly to the applet instance on that logical channel. If no applet is active, then only SELECT FILE commands for applet selection or MANAGE CHANNEL commands can be processed on that logical channel.

The mechanism for specifying the default applet instance for a logical channel is not defined in the Java Card API. It is a Java Card RE implementation detail and is left to the individual implementers.

4.2 Multiselectable Applets

Applets having the capability of being selected on multiple logical channels at the same time, or accepting other applets belonging to the same package being selected simultaneously, are referred to as *multiselectable* applets.

Note – All applets within a package shall be multiselectable or none shall be.

An *applet's context is active* when either an instance of the applet is already active, or when another applet instance from the same package is active. For more information about contexts see Section 6.1.2 "Contexts and Context Switching." An attempt to select an applet instance when the applet's context is active, is referred to as a *multiselection* attempt. If successful, multiselection occurs, and the applet instance becomes *multiselected*.

Multiselectable applets shall implement the `javacard.framework.MultiSelectable` interface. In case of multiselection, the applet instance will be informed by invoking its methods `MultiSelectable.select` and `MultiSelectable.deselect` during selection and deselection respectively.

When an applet instance not currently active is the first one selected in its package, its `Applet.select` method is called. Subsequent multiselections to this applet instance or selection of other applet instances in the same package shall result in a call to `MultiSelectable.select` method. This method is defined in the `MultiSelectable` interface. Its only purpose is to inform the applet instance that it will be multiselected. The applet instance may accept or reject a multiselection attempt.

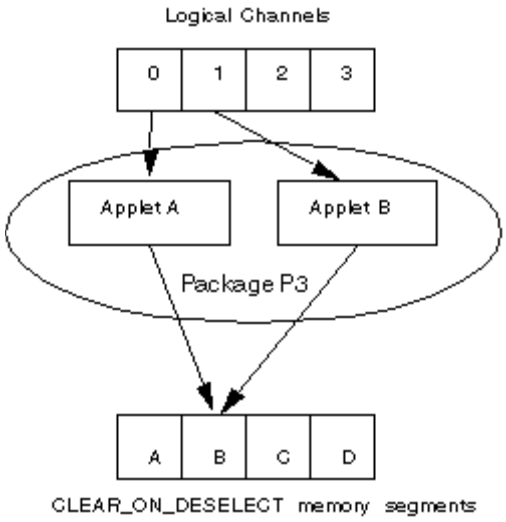
If a multiselection attempt is made on an applet which does not implement the `MultiSelectable` interface, the selection shall be rejected by the Java Card RE.

When a multiselected applet instance is deselected from one of the logical channels, the method `MultiSelectable.deselect` is called. Only when the multiselected applet instance is the last active applet instance in the applet's context, is its regular method `Applet.deselect` called.

There are two cases of multiselection:

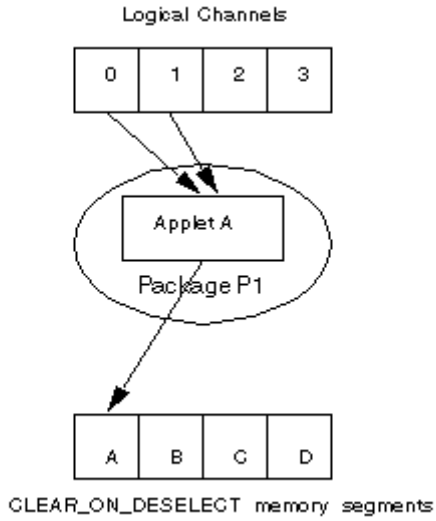
1. When two distinct applet instances from within the same package are multiselected, each applet instance shares the same `CLEAR_ON_DESELECT` memory transient segment. The applet instances share objects within the context firewall as well as their transient data. The Java Card RE shall not reset this `CLEAR_ON_DESELECT` transient objects until all applet instances within the package are deselected.(Refer to FIGURE 2.)

FIGURE 2 Different applet instances in same package



2. When the same applet instance is multiselected on two different logical channels simultaneously, it shares the `CLEAR_ON_DESELECT` memory segment space across logical channels. The Java Card RE shall not reset the `CLEAR_ON_DESELECT` transient objects until all applet instances within the package are deselected. (Refer to FIGURE 3.)

FIGURE 3 Same applet instance selected on multiple logical channels



In both cases of multiselection described above, the applet(s) must implement the `MultiSelectable` interface. If the applet(s) do not support this feature, then the selection must be rejected by the Java Card RE.

4.3 Forwarding APDU Commands To a Logical Channel

According to Section 5.4 of the *ISO 7816-4 Specification*, only APDU commands that contain the following encoding of the header CLA byte contain logical channel information:

`0x0X` or `0x8X` or `0x9X` or `0xAX`.

The X nibble is responsible for logical channels and secure messaging encoding. Only the two least significant bits (b2,b1*) of the nibble are used for logical channel encoding, which ranges from `0` to `3`. When an APDU command is received, the Java Card RE shall process it and determine whether or not the command has logical channel information. If logical channel information is encoded, then the card dispatches the APDU command to the appropriate logical channel. All other APDU commands are forwarded to the card's basic logical channel (logical channel `0`).

The Java Card RE always forwards the command “as is” to the appropriate applet instance. In particular, the Java Card RE does not clear the least significant bits (b2,b1*) of the CLA byte.

Note – An asterisk in the following sections indicates binary notation (%b) using bit numbering as in the ISO7816 specification. Most significant bit is b8. Least significant bit = b1.

4.4 Opening and Closing Logical Channels

According to Section 5.5.2 of the *ISO 7816-4 Specification*, there are two ways to open a logical channel in the smart card:

1. By selecting an applet instance on a new logical channel. This is accomplished by issuing a Applet SELECT FILE APDU command, and specifying in the lower two bits of the CLA byte the logical channel’s number (from 0 to 3). If this logical channel is currently closed, it shall be opened, and the specified applet instance shall be selected. (See Section 4.5.2 “Applet Selection with SELECT FILE.”)
2. By issuing a MANAGE CHANNEL OPEN APDU command. MANAGE CHANNEL commands are provided to open a logical channel from another logical channel, or to close a logical channel from another logical channel. (See Section 4.4.1.)

4.4.1 MANAGE CHANNEL Command Processing

The Java Card RE shall intercept all APDU messages coming into the card, perform card management functions (such as selecting or deselecting applet instances), and shall forward APDU messages to the appropriate applet instance. As part of its card management functions, the Java Card RE notifies applet instances about selection events (a function it performs by calling the applet instances’ `select` and `deselect` methods), and it initiates APDU processing within the currently selected applet instance by calling the applet instance’s `process` method.

With the addition of logical channels in Java Card platform, the Java Card RE includes a multichannel dispatching mechanism, as well as checks to ensure applet integrity during multi-channel operations. The Java Card RE must ensure that applets written to operate in a single logical channel environment operate consistently on a multiple logical channel smart card.

Java Card platform defines a class of APDU commands, called MANAGE CHANNEL commands. The following subsections summarize the functions the Java Card RE must perform by using MANAGE CHANNEL command processing:

MANAGE CHANNEL OPEN: Open a new logical channel from an already-open logical channel. Two variations of this command are supported:

- The Java Card RE selects the new logical channel specified in the command
- The Java Card RE automatically assigns a new logical channel.

MANAGE CHANNEL CLOSE: Close a specified logical channel from another open logical channel.

In addition, the SELECT FILE APDU command to select an applet instance is extended to specify a new or already opened logical channel on which the specified applet instance is to be selected.

In the following sections, the term *origin logical channel* is used to refer to the logical channel on which the command was received: that is, the two least significant bits (b2,b1*) of the CLA byte, as described in Section 4.3.

4.5 Applet Selection

There are two ways to select an applet instance in the Java Card platform: with a MANAGE CHANNEL OPEN command (Section 4.5.1), or with a SELECT FILE command, (See Section 4.5.2 "Applet Selection with SELECT FILE.")

The Java Card RE shall guarantee that an applet that is designed to run on any logical channel can be selected on any of the available logical channels on the card. The resources accessed by the applet instance must be the same, irrespective of the logical channel it is selected on.

4.5.1 Applet Selection with MANAGE CHANNEL OPEN

Upon receiving a MANAGE CHANNEL OPEN command, the Java Card RE shall run the following procedure:

1. The MANAGE CHANNEL OPEN command uses: CLA=%b000000cc* (where cc in the bits (b2,b1) denotes the origin logical channel: `0-3`), INS=`0x70` and P1=`0`. Two variants of this command are supported:
   - P2=`0` when the Java Card RE shall assign a new logical channel number
   - P2= the logical channel number specified.
     - If the MANAGE CHANNEL OPEN command has non-zero secure messaging bits (b4,b3*) in the CLA byte, the Java Card RE responds with status code `0x6882` (`SW_SECURE_MESSAGING_NOT_SUPPORTED`).

- If the MANAGE CHANNEL command is issued with P1 not equal to `0` or `0x80`, or if the unsigned value of P2 is greater than `3`, the Java Card RE responds with status code `0x6A81` (`SW_FUNC_NOT_SUPPORTED`).

2. If the origin logical channel is not open, the Java Card RE responds with status code `0x6881` (`SW_LOGICAL_CHANNEL_NOT_SUPPORTED`).

3. If the Java Card RE supports only the basic logical channel, the Java Card RE responds with status code `0x6881` (`SW_LOGICAL_CHANNEL_NOT_SUPPORTED`)

4. If the P2=`0` variant is used:
   - If the expected length value (Le) is not equal to `1`, the Java Card RE responds with status code `0x6C01` (`SW_CORRECT_LENGTH_00+0x01`).
   - If resources for the new logical channel are not available, the Java Card RE responds with status code `0x6A81` (`SW_FUNC_NOT_SUPPORTED`.)

5. If the P2 != `0` variant is used:

   If resources for the specified logical channel are not available or the logical channel is already open, the Java Card RE responds with status code `0x6A86` (`SW_INCORRECT_P1P2`).

6. The new logical channel is now open. This logical channel will be the *assigned channel* for the applet instance that will be selected on it.

7. Determine the applet instance to be selected on the new logical channel.
   - If the origin logical channel is the basic logical channel (logical channel `0`), then:
     - If a default applet instance for the new logical channel is defined, pick the default applet instance for that logical channel as the candidate for selection on the new logical channel.
     - Otherwise, set the Java Card RE state so that no applet is active on the new logical channel. The Java Card RE responds with status code `0x9000` and if the P2=`0` variant is used, 1 data byte containing the newly assigned logical channel number.
   - If the origin logical channel is not the basic logical channel:
     - If an applet instance is active on the origin logical channel, pick the applet instance as the candidate for selection on the new logical channel.
     - Otherwise, set the Java Card RE state so that no applet is active on the new logical channel. The Java Card RE responds with status code `0x9000` and if the P2=`0` variant is used, 1 data byte containing the newly assigned logical channel number.

8. If the candidate applet instance is not a multiselectable applet (as defined in Section 4.2) and the candidate applet's context is active, the Java Card RE shall close the new logical channel. The Java Card RE responds with status code `0x6985` (`SW_CONDITIONS_NOT_SATISFIED`).

9. Assign the `CLEAR_ON_DESELECT` transient memory segment for the new logical channel:

- If the applet's context is active, assign the `CLEAR_ON_DESELECT` transient memory segment associated with that context to this logical channel.
- Otherwise, assign a new (zero-filled) `CLEAR_ON_DESELECT` transient memory segment to this new logical channel.

10. Check whether the candidate applet instance accepts selection:
    - If the candidate applet's context is active, the Java Card RE shall set the candidate applet instance as the currently selected applet instance and call the `MultiSelectable.select` method, where the parameter `appInstAlreadyActive` is set to `true` if the same applet instance is already active on another logical channel. A context switch into the candidate applet instance's context occurs at this point. (For more details on contexts, see Section 6.1.2)
    - Otherwise, if the candidate applet's context is not active, the Java Card RE shall set the candidate applet instance as the currently selected applet instance and call the `Applet.select` method. A context switch into the candidate applet instance's context occurs at this point.
    - If the applet instance's select method throws an exception or returns false, then the Java Card RE closes the new logical channel. The Java Card RE responds with status code `0x6999` (`SW_APPLET_SELECT_FAILED`).
11. The Java Card RE responds with status code `0x9000` (and if the P2=`0` variant is used, 1 data byte containing the newly assigned logical channel number.)

Note – Unlike the SELECT FILE commands to select an applet instance, the MANAGE CHANNEL command is never forwarded to the applet instance.

4.5.2 Applet Selection with SELECT FILE

Upon receiving a SELECT FILE command, the Java Card RE shall run the following procedure:

1. The Applet SELECT FILE command uses: CLA=`%b000000cc`* (where cc in the bits (b2,b1*) specifies the logical channel to be selected: 0-3) and INS=`0xA4`.

   If the SELECT FILE command has non-zero secure messaging bits (b4,b3*) in the CLA byte, it is deemed not to be an Applet SELECT FILE command. The Java Card RE simply forwards the command to the active applet on the specified logical channel.
   - The Applet SELECT FILE command uses "Selection by DF name" with P1=`0x04`.
   - The Java Card RE shall support both :
     - selection by "exact DF name(AID)" with P2=`%b0000xx00` (b4,b3* are don't care) and

- the RFU variant described in ISO7816-4 specification with P2=`%b0001xx00` (b4,b3* are don't care).

- All other partial DF name SELECT FILE options (b2,b1* variants) are Java Card RE implementation-dependent.
- All file control information options codes (b4,b3*) shall be supported by the Java Card RE and interpreted and processed by the applet instance itself.

2. If resources for the specified logical channel (in cc) are not available, the Java Card RE responds with status code `0x6881` (`SW_LOGICAL_CHANNEL_NOT_SUPPORTED`)

3. If the specified logical channel is not open, it is now opened and the Java Card RE state is set so that no applet is active on this new logical channel. The specified logical channel will be the *assigned channel* for the applet instance that will be active on it.

4. The Java Card RE searches the internal applet table which lists all successfully installed applet instances on the card for an applet instance with a matching AID. If a matching applet instance is found, it is picked as the candidate applet instance. Otherwise, if no AID match is found:
   - If there is no active applet instance on the specified logical channel, the Java Card RE responds with status code `0x6999` (`SW_APPLET_SELECT_FAILED`).
   - Otherwise, the active applet instance on this logical channel is set as the currently selected applet instance and the SELECT FILE command is forwarded to that applet instance's process method. A context switch into the applet instance's context occurs at this point (See Section 6.1.1). Applets may use the SELECT FILE command for their own internal processing. Upon return from the applet's process method, the Java Card RE sends the applet instance's response as the response to the SELECT FILE command.

5. If the candidate applet instance is not a multiselectable applet, and the candidate applet's context is active, the logical channel remains open and the Java Card RE records an error response status code of `0x6985` (`SW_CONDITIONS_NOT_SATISFIED`). Prior to sending the response code, if there is an active applet instance on the logical channel, then the Java Card RE may optionally deselect the applet instance, as described in Section 4.6 "Applet Deselection" and set the state so that no applet is active on the specified logical channel.

6. Assign the `CLEAR_ON_DESELECT` transient memory segment for the new logical channel:
   - If any applet instance from the same package as that of the candidate applet instance is active on another logical channel, assign the same `CLEAR_ON_DESELECT` transient memory segment to this logical channel.
   - Otherwise, assign a different (zero-filled) `CLEAR_ON_DESELECT` transient memory segment to this new logical channel.

7. Check whether the candidate applet instance accepts selection:

- If the candidate applet's context is active, the Java Card RE shall set the candidate applet instance as the currently selected applet instance and call the `MultiSelectable.select(appInstAlreadyActive)` method, where the parameter `appInstAlreadyActive` is set to true if the same applet instance is already active on another logical channel. A context switch into the candidate applet instance's context occurs at this point (See Section 6.1.2).
- Otherwise, if the candidate applet's context is not active, the Java Card RE shall set the candidate applet instance as the currently selected applet instance and call the `Applet.select` method. A context switch into the candidate applet instance's context occurs at this point.
- If the applet instance's select method throws an exception or returns `false`, then the Java Card RE state is set so that no applet is active on the specified logical channel. The logical channel remains open, and the Java Card RE responds with status code `0x6999` (`SW_APPLET_SELECT_FAILED`).

8. The Java Card RE shall set the candidate applet instance as the currently selected applet instance and call the `Applet.process` method with the SELECT FILE APDU as the input parameter. A context switch occurs into the applet instance's context at this point. Upon return from the applet instance's process method, the Java Card RE sends the applet instance's response as the response to the SELECT FILE command.

Note –
If the SELECT FILE command does not conform to the exact format of an Applet SELECT FILE command described in item 1 above or if there is no matching AID, the SELECT FILE command is forwarded to the active applet instance (if any) on that logical channel for processing as a normal applet APDU command.

If there is a matching AID and the SELECT FILE command fails, the Java Card RE always sets the state in which no applet is active on that logical channel.

If the matching AID is the same as the active applet instance on the specified logical channel, the Java Card RE still goes through the process of deselecting the applet instance and then selecting it. Reselection could fail, leaving the card in a state in which no applet is active on that logical channel.

4.6 Applet Deselection

An applet instance is deselected either upon receipt of a MANAGE CHANNEL CLOSE command, or as a result of a SELECT FILE command that selects a different (or the same) applet instance on the specified logical channel.

In either case, when an applet instance is deselected the following procedure shall be followed by the Java Card RE:

- If the applet instance to be deselected is active on more than one logical channel, or another applet instance from the same package is also active, the Java Card RE sets the currently selected applet instance to be the applet instance being deselected, and calls its `MultiSelectable.deselect(appInstStillActive)` method, where the `appInstStillActive` parameter is set to `true` if the same applet instance is still active on another logical channel. A context switch occurs into the applet instance's context at this point (See Section 6.1.2).
- Otherwise, the Java Card RE sets the currently selected applet instance to be the applet instance being deselected, and calls its `Applet.deselect` method. Upon return or uncaught exception, the Java Card RE clears the fields of all `CLEAR_ON_DESELECT` transient objects in the context of deselected applet instance.

Note – Note that the deselection is always successful even if the applet instance throws an exception from within the `deselect` method.

4.6.1 MANAGE CHANNEL CLOSE Command

Upon receiving a MANAGE CHANNEL CLOSE command, the Java Card RE shall run the following procedure:

1. The MANAGE CHANNEL CLOSE command uses: CLA=`%b000000cc`* (where cc in the bits (b2,b1) denotes the origin logical channel: `0-3`), INS=`0x70`, P1=`0x80` and P2 specifies the logical channel to be closed.
   - If the MANAGE CHANNEL CLOSE command has non-zero secure messaging bits (b4,b3) in the CLA byte, the Java Card RE responds with status code `0x6882` (`SW_SECURE_MESSAGING_NOT_SUPPORTED`).
   - If the MANAGE CHANNEL command is issued with P1 not equal `0` or `0x80`, the Java Card RE responds with status code `0x6A81` (`SW_FUNC_NOT_SUPPORTED`).
2. If the origin logical channel is not open, the Java Card RE responds with status code `0x6881` (`SW_LOGICAL_CHANNEL_NOT_SUPPORTED`).
3. If the Java Card RE supports only the basic logical channel, the Java Card RE responds with status code `0x6881` (`SW_LOGICAL_CHANNEL_NOT_SUPPORTED`).
4. If the specified logical channel to close is the *basic logical channel* (logical channel `0`) or the specified logical channel number is greater than `3`, the Java Card RE responds with status code `0x6A81` (`SW_FUNC_NOT_SUPPORTED`).
5. If the specified logical channel to close is currently open, deselect the active applet instance (if any) on the specified logical channel as described above in Section 4.6. The specified logical channel is now closed. The Java Card RE responds with status code `0x9000`.

6. Otherwise, if the specified logical channel is closed or not available, the Java Card RE responds with warning status code `0x6200` (`SW_WARNING_STATE_UNCHANGED`).

4.7 Other Command Processing

When an APDU other than a SELECT FILE or MANAGE CHANNEL command is received, the logical channel to be used for dispatching the command is based on the CLA byte as described in Section 4.3 “Forwarding APDU Commands To a Logical Channel.”

When the Java Card RE receives an APDU other than a SELECT FILE or MANAGE CHANNEL command with either:

- an unsupported logical channel number in the CLA byte, or
- an unopened logical channel number in the CLA byte

it shall respond to the APDU with status code `0x6881` (`SW_LOGICAL_CHANNEL_NOT_SUPPORTED`).

If there is no active applet instance on the logical channel to be used for dispatching the command, the Java Card RE shall respond to the APDU with status code `0x6999` (`SW_APPLET_SELECT_FAILED`).

When an APDU other than a Applet SELECT FILE or a MANAGE CHANNEL command is received, and there is an active applet instance on the logical channel to be used for dispatching the command, the Java Card RE sets the active applet instance on the origin channel as the currently selected applet instance and invokes the process method passing the APDU as a parameter. This causes a context switch from the Java Card RE context into the currently selected applet instance’s context (For more information on contexts see Section 6.1.2 “Contexts and Context Switching.”) When the `process` method exits, the VM switches back to the Java Card RE context. The Java Card RE sends the response APDU and waits for the next command APDU.

Note that the Java Card RE dispatches the APDU command “as is” to the applet instance for processing via the `process` method. Therefore, the CLA byte in the command header will contain in its least significant bits the origin channel number. An applet designed to run on any logical channel needs to mask out these two bits before checking for specific values.

CHAPTER 5

Transient Objects

Applets sometimes require objects that contain temporary (transient) data that need not be persistent across CAD sessions. Java Card platform does not support the Java-language keyword `transient`. However, Java Card technology provides methods to create transient arrays with primitive components or references to `Object`.

Note – In this section, the term *field* is used to refer to the *component* of an array object also.

The term "transient object" is a misnomer. It can be incorrectly interpreted to mean that the object itself is transient. However, only the *contents* of the fields of the object (except for the length field) have a transient nature. As with any other object in the Java programming language, transient objects within the Java Card platform exist as long as they are referenced from:

- The stack
- Local variables
- A class static field
- A field in another existing object

A transient object within the Java Card platform has the following required behavior:

- The fields of a transient object shall be *cleared* to the field's default value (zero, `false`, or `null`) at the occurrence of certain events (see Section 5.1 "Events That Clear Transient Objects").
- For security reasons, the fields of a transient object shall never be stored in a "persistent memory technology." Using current smart card technology as an example, the contents of transient objects can be stored in RAM, but never in EEPROM. The purpose of this requirement is to allow transient objects to be used to store session keys.
- Writes to the fields of a transient object shall not have a performance penalty. (Using current smart card technology as an example, the contents of transient objects can be stored in RAM, while the contents of persistent objects can be stored in EEPROM. Typically, RAM technology has a much faster write cycle time than EEPROM.)

- Writes to the fields of a transient object shall not be affected by "transactions." That is, an `abortTransaction` will never cause a field in a transient object to be restored to a previous value.

This behavior makes transient objects ideal for small amounts of temporary applet data that is frequently modified, but that need not be preserved across CAD or select sessions.

5.1 Events That Clear Transient Objects

Persistent objects are used for maintaining states that shall be preserved across card resets. When a transient object is created, one of two events is specified that causes its fields to be cleared. `CLEAR_ON_RESET` transient objects are used for maintaining states that shall be preserved across applet selections, but not across card resets. `CLEAR_ON_DESELECT` transient objects are used for maintaining states that must be preserved while an applet is selected, but not across applet selections or card resets.

Details of the two clear events are as follows:

- `CLEAR_ON_RESET`—the object's fields (except for the length field) are cleared when the card is reset. When a card is powered on, this also causes a card reset.

Note – It is not necessary to clear the fields of transient objects before power is removed from a card. However, it is necessary to guarantee that the previous contents of such fields cannot be recovered once power is lost.

- `CLEAR_ON_DESELECT`—the object's fields (except for the length field) are cleared whenever the applet is deselected and no other applets from the same package are active on the card. Because a card reset implicitly deselects the currently selected applet, the fields of `CLEAR_ON_DESELECT` objects are also cleared by the same events specified for `CLEAR_ON_RESET`.

The currently selected applet is explicitly deselected (its `deselect` method is called) only when a `SELECT FILE` command or MANAGE CHANNEL CLOSE command is processed. The currently selected applet is deselected and then the fields of all `CLEAR_ON_DESELECT` transient objects owned by the applet are cleared if no other applets from the same package are active on the card, regardless of whether the `SELECT FILE` command:

- Fails to select an applet.
- Selects a different applet.
- Reselects the same applet.

CHAPTER 6

Applet Isolation and Object Sharing

Any implementation of the Java Card RE shall support isolation of contexts and applets. Isolation means that one applet can not access the fields or objects of an applet in another context unless the other applet explicitly provides an interface for access. The Java Card RE mechanisms for applet isolation and object sharing are detailed in the sections below.

6.1 Applet Firewall

The *applet firewall* within Java Card technology is runtime-enforced protection and is separate from the Java technology protections. The Java language protections still apply to Java Card applets. The Java language ensures that strong typing and protection attributes are enforced.

Applet firewalls are always enforced in the Java Card VM. They allow the VM to automatically perform additional security checks at runtime.

6.1.1 Firewall Protection

The Java Card technology-based firewall ("Java Card firewall") provides protection against the most frequently anticipated security concern: developer mistakes and design oversights that might allow sensitive data to be "leaked" to another applet. An applet may be able to obtain an object reference from a publicly accessible location. However, if the object is owned by an applet protected by its own firewall, then the requesting applet must satisfy certain access rules before it can use the reference to access the object.

The firewall also provides protection against incorrect code. If incorrect code is loaded onto a card, the firewall still protects objects from being accessed by this code.

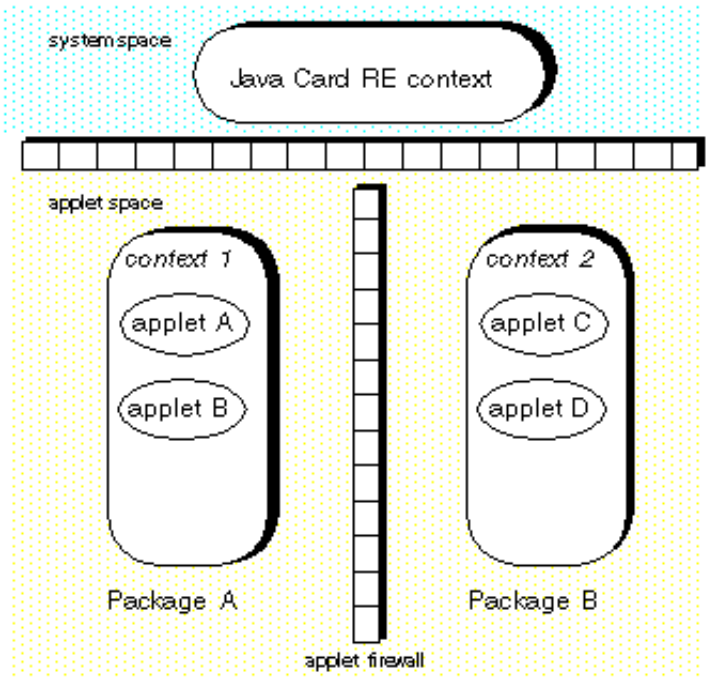
The *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1* specifies the basic minimum protection requirements of contexts and firewalls because the features described in this document are not transparent to the applet developer. Developers shall be aware of the behavior of objects, APIs, and exceptions related to the firewall.

Java Card RE implementers are free to implement additional security mechanisms beyond those of the applet firewall, as long as these mechanisms are transparent to applets and do not change the externally visible operation of the VM.

6.1.2 Contexts and Context Switching

Firewalls essentially partition the Java Card platform's object system into separate protected object spaces called *contexts*. These are illustrated in FIGURE 4. The firewall is the boundary between one context and another. The Java Card RE shall allocate and manage a *context* for each Java API package containing applets[1]. All applet instances within a single Java API package share the same context. There is no firewall between individual applet instances within the same package; that is, an applet instance can freely access objects belonging to another applet instance that resides in the same package.

FIGURE 4 Contexts within the Java Card Platform's Object System



1. Note that a library package is not assigned a separate context. Objects from a library package belong to the context of the creating applet instance.

In addition, the Java Card RE maintains its own *Java Card RE context*. This context is much like the context of an applet, but it has special system privileges so that it can perform operations that are denied to contexts of applets. For example, access from the Java Card RE context to any applet instance's context is allowed, but the converse, access from an applet instance's context to the Java Card RE context, is prohibited by the firewall.

6.1.2.1 Active Contexts in the VM

At any point in time, there is only one *active context* within the VM. (This is called the *currently active context*.) This can be either the Java Card RE context or an applet's context. All bytecodes that access objects are checked at *runtime* against the currently active context in order to determine if the access is allowed. A `java.lang.SecurityException` is thrown when an access is disallowed.

6.1.2.2 Context Switching in the VM

If access is allowed, the VM determines if a *context switch* is required. A context switch occurs when certain well-defined conditions, as described in Section 6.2.8, are met during the execution of invoke-type bytecodes. For example, a context switch may be caused by an attempt to access a shareable object that belongs to an applet instance that resides in a different package. The result of a context switch is a new currently active context.

During a context switch, the previous context and object owner information is pushed on an internal VM stack, a new context becomes the currently active context, and the invoked method executes in this new context. Upon exit from that method the VM performs a restoring context switch. The original context (of the caller of the method) is popped from the stack and is restored as the currently active context. Context switches can be nested. The maximum depth depends on the amount of VM stack space available.

Most method invocations in Java Card technology do not cause a context switch. For example, a context switch is unnecessary when an attempt is made to access an object that belongs to an applet instance that resides in the same package. Context switches only occur during invocation of and return from certain methods, as well as during exception exits from those methods (see Section 6.2.8).

Further details of contexts and context switching are provided in later sections of this chapter.

6.1.3 Object Ownership

Any given object in the Java Card platform's object space has a context and an owner associated with it. When a new object is created, it is associated with the currently active context. But the object is *owned* by the applet instance within the currently active context when the object is instantiated. An object can be owned by an applet instance, or by the Java Card RE.

The combined rules of context and object ownership within the firewall are:

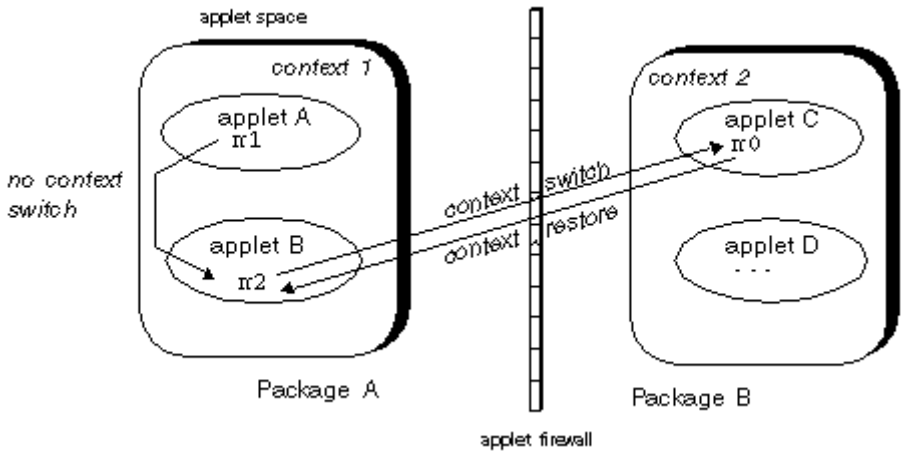
- Every applet instance belongs to a context. All applet instances from the same package belong to the same context.
- Every object is owned by an applet instance (or the Java Card RE). An applet instance is identified by its AID. When executing in an instance method of an object (or a static class method called from within), the object's owner must be in the currently active context.

For example, assume that applets A and B are in the same package, and applet C is in another package. A and B therefore belong to the same context: 1. C belongs to a different context: 2. For an illustration of this situation, see FIGURE 5.

If Context 1 is the currently active context, and a method `m1` in an object owned by applet A is invoked, no context switch occurs. If method `m1` invokes a method `m2` in an object owned by applet B, again no context switch occurs (in spite of the object "owner" change), and no firewall restrictions apply.

However, if the method `m2` now calls a method `m0` in an object owned by applet C, firewall restrictions apply and, if access is allowed, a context switch shall occur. Upon return to method `m2` from the method `m0`, the context of applet B is restored.

FIGURE 5 Context Switching and Object Access



To emphasize a few points:

- when the m1 method in the object owned by applet A calls the method m2 in the object owned by applet B, the context does not change but the owner of the object does change. If the JCSystem.getAID method is called from method m2 within context 1, the AID of Applet B is returned.
- when method m2 calls method m0 in an object owned by applet C, applet B is the owner of the object when the context switches from 1 to 2. Therefore, if the JCSystem.getAID method is called from method m0 within context 2, the AID of applet C shall be returned. If the JCSystem.getPreviousContextAID method is called, the AID of applet B shall be returned.
- when the JCSystem.getAID method is called from method m2 after the return from method m0 in context 2, the AID of applet B is returned. However, if the JCSystem.getPreviousContextAID method is called, the AID of the applet which called into context 1 (or null if Java Card RE) is returned and not the AID of applet C.

6.1.4 Object Access

In general, an object can only be *accessed* by its owning context, that is, when the owning context is the currently active context. The firewall prevents an object from being accessed by another applet in a different context.

In implementation terms, each time an object is accessed, the object's owner context is compared to the currently active context. If these do not match, the access is not performed and a SecurityException is thrown.

An object is accessed when one of the following bytecodes is executed using the object's reference:

```
getfield, putfield, invokevirtual, invokeinterface,
   athrow, <T>aload, <T>astore, arraylength, checkcast, instanceof
```

<T> refers to the various types of array bytecodes, such as baload, sastore, etc.

This list includes any special or optimized forms of these bytecodes implemented in the Java Card VM, such as getfield_b, getfield_s_this, etc.

6.1.5 Transient Objects and Contexts

Transient objects of CLEAR_ON_RESET type behave like persistent objects in that they can be accessed only when the currently active context is the object's owning context (the currently active context at the time when the object was created).

Transient objects of CLEAR_ON_DESELECT type can only be created or accessed when the currently active context is the context of the currently selected applet. If any of the makeTransient factory methods of JCSystem class are called to create a CLEAR_ON_DESELECT type transient object when the currently active context is not the context of the currently selected applet (even if the attempting context is that of an active applet instance on another logical channel), the method shall

throw a `java.lang.SystemException` with reason code of `ILLEGAL_TRANSIENT`. If an attempt is made to access a transient object of `CLEAR_ON_DESELECT` type when the currently active context is not the context of the currently selected applet (even if the attempting context is that of an active applet instance on another logical channel), the Java Card RE shall throw a `java.lang.SecurityException`.

Applets that are part of the same package share the same context. Every applet instance from a package shares all its object instances with all other instances from the same package. (This includes transient objects of both `CLEAR_ON_RESET` type and `CLEAR_ON_DESELECT` type owned by these applet instances.)

The transient objects of `CLEAR_ON_DESELECT` type owned by any applet instance in the same package shall be accessible when any of the applet instances is the currently selected applet.

6.1.6 Static Fields and Methods

Instances of classes—objects—are owned by contexts; classes themselves are not. There is no runtime context check that can be performed when a class static field is accessed. Neither is there a context switch when a static method is invoked. (Similarly, `invokespecial` causes no context switch.)

Public static fields and public static methods are accessible from any context: static methods execute in the same context as their caller.

Objects referenced in static fields are just regular objects. They are owned by whomever created them and standard firewall access rules apply. If it is necessary to share them across multiple contexts, then these objects need to be *Shareable Interface Objects* (SIOs). (See Section 6.2.4 below.)

Of course, the conventional Java technology protections are still enforced for static fields and methods. In addition, when applets are installed, the Installer verifies that each attempt to link to an external static field or method is permitted. Installation and specifics about linkage are beyond the scope of this specification.

6.1.6.1 Optional static access checks

The Java Card RE may perform optional runtime checks that are redundant with the constraints enforced by a verifier. A Java Card VM may detect when code violates fundamental language restrictions, such as invoking a private method in another class, and report or otherwise address the violation.

6.2 Object Access Across Contexts

The applet firewall confines an applets actions to its designated context. To enable applets to interact with each other and with the Java Card RE, some well-defined yet secure mechanisms are provided so one context can access an object belonging to another context.

These mechanisms are provided in the Java Card API and are discussed in the following sections:

- Java Card RE Entry Point Objects
- Global Arrays
- Java Card RE Privileges
- Shareable Interfaces

6.2.1 Java Card RE Entry Point Objects

Secure computer systems must have a way for non-privileged user processes (that are restricted to a subset of resources) to request system services performed by privileged "system" routines.

In the Java Card API, this is accomplished using *Java Card RE Entry Point Objects*. These are objects owned by the Java Card RE context, but they have been flagged as containing entry point methods.

The firewall protects these objects from access by applets. The entry point designation allows the methods of these objects to be invoked from any context. When that occurs, a context switch to the Java Card RE context is performed. These methods are the gateways through which applets request privileged Java Card RE system services. The requested service is performed by the entry point method after verifying that the method parameters are within bounds and all objects passed in as parameters are accessible from the caller's context.

There are two categories of Java Card RE Entry Point Objects:

- Temporary Java Card RE Entry Point Objects

  Like all Java Card RE Entry Point Objects, methods of temporary Java Card RE Entry Point Objects can be invoked from any context. However, references to these 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 re-use.

  The APDU object and all Java Card RE owned exception objects are examples of temporary Java Card RE Entry Point Objects.
- Permanent Java Card RE Entry Point Objects

Like all Java Card RE Entry Point Objects, methods of permanent Java Card RE Entry Point Objects can be invoked from any context. Additionally, references to these objects can be stored and freely re-used.

Java Card RE owned AID instances are examples of permanent Java Card RE Entry Point Objects.

The Java Card RE is responsible for:

- Determining what privileged services are provided to applets.
- Defining classes containing the entry point methods for those services.
- Creating one or more object instances of those classes.
- Designating those instances as Java Card RE Entry Point Objects.
- Designating Java Card RE Entry Point Objects as temporary or permanent.
- Making references to those objects available to applets as needed.

Note – Only the *methods* of these objects are accessible through the firewall. The fields of these objects are still protected by the firewall and can only be accessed by the Java Card RE context.

Only the Java Card RE itself can designate Entry Point Objects and whether they are temporary or permanent. Java Card RE implementers are responsible for implementing the mechanism by which Java Card RE Entry Point Objects are designated and how they become temporary or permanent.

6.2.2 Global Arrays

The global nature of some objects requires that they be accessible from any context. The firewall would ordinarily prevent these objects from being used in a flexible manner. The Java Card VM allows an object to be designated as *global*.

All global arrays are temporary global array objects. These objects are owned by the Java Card RE context, but can be accessed from any context. However, references to these 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 re-use.

For added security, only arrays can be designated as global and only the Java Card RE itself can designate global arrays. Because applets cannot create them, no API methods are defined. Java Card RE implementers are responsible for implementing the mechanism by which global arrays are designated.

At the time of publication of this specification, the only global arrays required in the Java Card API are the APDU buffer and the byte array input parameter (`bArray`) to the applet's `install` method.

Note – Because of the global status of the APDU buffer, the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* specifies that this buffer is cleared to zeroes whenever an applet is selected, before the Java Card RE accepts a new APDU command. This is to prevent an applet's potentially sensitive data from being "leaked" to another applet via the global APDU buffer. The APDU buffer can be accessed from a shared interface object context and is suitable for passing data across different contexts. The applet is responsible for protecting secret data that may be accessed from the APDU buffer.

6.2.3 Java Card RE Privileges

Because it is the "system" context, the Java Card RE context has a special privilege. It can invoke a method of any object on the card. For example, assume that object X is owned by applet A. Normally, only the context of A can access the fields and methods of X. But the Java Card RE context is allowed to invoke any of the methods of X. During such an invocation, a context switch occurs from the Java Card RE context to the context of the applet that owns X.

Again, because it is the "system" context, the Java Card RE context can access fields and components of any object on the card including `CLEAR_ON_DESELECT` transient objects owned by the currently selected applet.

Note – The Java Card RE can access both *methods* and *fields* of X. Method access is the mechanism by which the Java Card RE enters the context of an applet. Although the Java Card RE could invoke any method through the firewall, it shall only invoke the `select`, `process`, `deselect`, and `getShareableInterfaceObject` (see Section 6.2.7.1) methods defined in the `Applet` class, and methods on the objects passed to the API as parameters.

The Java Card RE context is the currently active context when the VM begins running after a card reset. The Java Card RE context is the "root" context and is always either the currently active context or the bottom context saved on the stack.

6.2.4 Shareable Interfaces

Shareable interfaces are a feature in the Java Card API to enable applet interaction. A shareable interface defines a set of shared interface methods. These interface methods can be invoked from one context even if the object implementing them is owned by an applet in another context.

In this specification, an object instance of a class implementing a shareable interface is called a *Shareable Interface Object* (SIO).

To the owning context, the SIO is a normal object whose fields and methods can be accessed. To any other context, the SIO is an instance of the shareable interface, and only the methods defined in the shareable interface are accessible. All other fields and methods of the SIO are protected by the firewall.

Shareable interfaces provide a secure mechanism for inter-applet communication, as follows:

6.2.4.1 Server applet A builds a Shareable Interface Object

1. To make an object available for sharing with another applet in a different context, applet A first defines a shareable interface, SI. A shareable interface extends the interface `javacard.framework.Shareable`. The methods defined in the shareable interface, SI, represent the services that applet A makes accessible to other applets.
2. Applet A then defines a class C that implements the shareable interface SI. C implements the methods defined in SI. C may also define other methods and fields, but these are protected by the applet firewall. Only the methods defined in SI are accessible to other applets.
3. Applet A creates an object instance O of class C. O belongs to applet A, and the firewall allows A to access any of the fields and methods of O.

6.2.4.2 Client applet B obtains the Shareable Interface Object

1. To access applet A's object O, applet B creates an object reference SIO of type SI.
2. Applet B invokes a special method (`JCSystem.getAppletShareableInterfaceObject`, described in Section 6.2.7.2) to request a shared interface object reference from applet A.
3. Applet A receives the request and the AID of the requester (B) via `Applet.getShareableInterfaceObject`, and determines whether it will share object O with applet B. A's implementation of the `getShareableInterfaceObject` method executes in A's context.
4. If applet A agrees to share with applet B, A responds to the request with a reference to O. As this reference is returned as type `Shareable`, none of the fields or methods of O are visible.
5. Applet B receives the object reference from applet A, casts it to the interface type SI, and stores it in object reference variable SIO. Even though SIO actually refers to A's object O, SIO is an interface of type SI. Only the shareable interface methods defined in SI are visible to B. The firewall prevents the other fields and methods of O from being accessed by B.

In the above sequence, applet B initiates communication with applet A using the special system method in the `JCSystem` class to request a Shareable Interface Object from applet A. Once this communication is established, applet B can obtain

other Shareable Interface Objects from applet A using normal parameter passing and return mechanisms. It can also continue to use the special `JCSystem` method described above to obtain other Shareable Interface Objects.

6.2.4.3 Client applet B requests services from applet A

1. Applet B can request service from applet A by invoking one of the shareable interface methods of SIO. During the invocation the Java Card VM performs a context switch. The original currently active context (B) is saved on a stack and the context of the owner (A) of the actual object (O) becomes the new currently active context. A's implementation of the shareable interface method (SI method) executes in A's context.
2. The SI method can find out the AID of its client (B) via the `JCSystem.getPreviousContextAID` method. This is described in Section 6.2.5. The method determines whether or not it will perform the service for applet B.
3. Because of the context switch, the firewall allows the SI method to access all the fields and methods of object O and any other object in the context of A. At the same time, the firewall prevents the method from accessing non-shared objects in the context of B.
4. The SI method can access the parameters passed by B and can provide a return value to B.
5. During the return, the Java Card VM performs a restoring context switch. The original currently active context (B) is popped from the stack, and again becomes the currently active context.
6. Because of the context switch, the firewall again allows B to access any of its objects and prevents B from accessing non-shared objects in the context of A.

6.2.5 Determining the Previous Context

When an applet calls `JCSystem.getPreviousContextAID`, the Java Card RE shall return the instance AID of the applet instance active at the time of the last context switch.

6.2.5.1 The Java Card RE Context

The Java Card RE context does not have an AID. If an applet calls the `getPreviousContextAID` method when the context of the applet was entered directly from the Java Card RE context, this method returns `null`.

If the applet calls `getPreviousContextAID` from a method that may be accessed either from within the applet itself or when accessed via a shareable interface from an external applet, it shall check for `null` return before performing caller AID authentication.

6.2.6 Shareable Interface Details

A shareable interface is simply one that extends (either directly or indirectly) the *tagging* interface `javacard.framework.Shareable`. This `Shareable` interface is similar in concept to the Remote interface used by the RMI facility, in which calls to the interface methods take place across a local/remote boundary.

6.2.6.1 The Java Card API Shareable Interface

Interfaces extending the `Shareable` tagging interface have this special property: calls to the interface methods take place across Java Card platform's applet firewall boundary via a context switch.

The `Shareable` interface serves to identify all shared objects. Any object that needs to be shared through the applet firewall shall directly or indirectly implement this interface. Only those methods specified in a shareable interface are available through the firewall.

Implementation classes can implement any number of shareable interfaces and can extend other shareable implementation classes.

Like any Java platform interface, a shareable interface simply defines a set of service methods. A service provider class declares that it "implements" the shareable interface and provides implementations for each of the service methods of the interface. A service client class accesses the services by obtaining an object reference, casting it to the shareable interface type, and invoking the service methods of the interface.

The shareable interfaces within the Java Card technology shall have the following properties:

- When a method in a shareable interface is invoked, a context switch occurs to the context of the object's owner.
- When the method exits, the context of the caller is restored.
- Exception handling is enhanced so that the currently active context is correctly restored during the stack frame unwinding that occurs as an exception is thrown.

6.2.7 Obtaining Shareable Interface Objects

Inter-applet communication is accomplished when a client applet invokes a shareable interface method of a SIO belonging to a server applet. For this to work, there must be a way for the client applet to obtain the SIO from the server applet in the first place. The Java Card RE provides a mechanism to make this possible. The `Applet` class and the `JCSystem` class provide methods to enable a client to request services from the server.

6.2.7.1 The Method Applet.getShareableInterfaceObject(AID, byte)

This method is implemented by the server applet instance. It shall be called by the Java Card RE to mediate between a client applet that requests to use an object belonging to another applet, and the server applet that makes its objects available for sharing.

The default behavior shall return `null`, which indicates that an applet does not participate in inter-applet communication.

A server applet that is intended to be invoked from another applet needs to override this method. This method should examine the `clientAID` and the parameter. If the `clientAID` is not one of the expected AIDs, the method should return `null`. Similarly, if the parameter is not recognized or if it is not allowed for the `clientAID`, then the method also should return `null`. Otherwise, the applet should return an SIO of the shareable interface type that the client has requested.

The server applet need not respond with the same SIO to all clients. The server can support multiple types of shared interfaces for different purposes and use `clientAID` and `parameter` to determine which kind of SIO to return to the client.

6.2.7.2 The Method JCSystem.getAppletShareableInterfaceObject

The `JCSystem` class contains the method `getAppletShareableInterfaceObject`, which is invoked by a client applet to communicate with a server applet.

The Java Card RE shall implement this method to behave as follows:

1. The Java Card RE searches its internal applet table which lists all successfully installed applets on the card for one with `serverAID`. If not found, `null` is returned.
2. The Java Card RE invokes this applet's `getShareableInterfaceObject` method, passing the `clientAID` of the caller and the parameter.
3. A context switch occurs to the server applet, and its implementation of `getShareableInterfaceObject` proceeds as described in the previous section. The server applet returns a SIO (or `null`).

4. `getAppletShareableInterfaceObject` returns the same SIO (or `null`) to its caller.

For enhanced security, the implementation shall make it impossible for the client to tell which of the following conditions caused a `null` value to be returned:

- The `serverAID` was not found.
- The server applet does not participate in inter-applet communication.
- The server applet does not recognize the `clientAID` or the `parameter`.
- The server applet won't communicate with this client.
- The server applet won't communicate with this client as specified by the `parameter`.

6.2.8 Class and Object Access Behavior

A static class field is *accessed* when one of the following Java language bytecodes is executed:

`getstatic, putstatic`

An object is *accessed* when one of the following Java language bytecodes is executed using the object's reference:

`getfield, putfield, invokevirtual, invokeinterface, athrow, <T>aload, <T>astore, arraylength, checkcast, instanceof`

`<T>` refers to the various types of array bytecodes, such as `baload`, `sastore`, etc.

This list also includes any special or optimized forms of these bytecodes that can be implemented in the Java Card VM, such as `getfield_b`, `getfield_s_this`, etc.

Prior to performing the work of the bytecode as specified by the Java VM, the Java Card VM will perform an *access check* on the referenced object. If access is denied, then a `java.lang.SecurityException` is thrown.

The access checks performed by the Java Card VM depend on the type and owner of the referenced object, the bytecode, and the currently active context. They are described in the following sections.

6.2.8.1 Accessing Static Class Fields

Bytecodes:

`getstatic, putstatic`

- If the Java Card RE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is `putstatic` and the field being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE Entry Point Object or a global array, then access is denied.
- Otherwise, access is allowed.

6.2.8.2 Accessing Array Objects

Bytecodes:

`<T>aload, <T>astore, arraylength, checkcast, instanceof`

- If the Java Card RE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is `aastore` and the component being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE Entry Point Object or a global array, then access is denied.
- Otherwise, if the array is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the array is designated global, then access is allowed.
- Otherwise, access is denied.

6.2.8.3 Accessing Class Instance Object Fields

Bytecodes:

`getfield, putfield`

- If the Java Card RE is the currently active context, then access is allowed.
- Otherwise, if the bytecode is `putfield` and the field being stored is a reference type and the reference being stored is a reference to a temporary Java Card RE Entry Point Object or a global array, then access is denied.
- Otherwise if the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, access is denied.

6.2.8.4 Accessing Class Instance Object Methods

Bytecodes:

`invokevirtual`

- If the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the object is designated a Java Card RE Entry Point Object, then access is allowed. Context is switched to the object owner's context (shall be Java Card RE).
- Otherwise, if Java Card RE is the currently active context, then access is allowed. Context is switched to the object owner's context.
- Otherwise, access is denied.

6.2.8.5 Accessing Standard Interface Methods

Bytecodes:

`invokeinterface`

- If the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the object is designated a Java Card RE Entry Point Object, then access is allowed. Context is switched to the object owner's context (shall be Java Card RE).
- Otherwise, if the Java Card RE is the currently active context, then access is allowed. Context is switched to the object owner's context.
- Otherwise, access is denied.

6.2.8.6 Accessing Shareable Interface Methods

Bytecodes:

`invokeinterface`

- If the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the object is owned by a non-multiselectable applet instance which is not in the context of the currently selected applet instance, and which is active on another logical channel, access is denied. (See Section 4.2)
- Otherwise, if the object's class implements a `Shareable` interface, and if the interface being invoked extends the `Shareable` interface, then access is allowed. Context is switched to the object owner's context.
- Otherwise, if the Java Card RE is the currently active context, then access is allowed. Context is switched to the object owner's context.
- Otherwise, access is denied.

6.2.8.7 Throwing Exception Objects

Bytecodes:

`athrow`

- If the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the object is designated a Java Card RE Entry Point Object, then access is allowed.
- Otherwise, if the Java Card RE is the currently active context, then access is allowed.
- Otherwise, access is denied.

6.2.8.8 Accessing Classes

Bytecodes:

`checkcast, instanceof`

- If the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the object is designated a Java Card RE Entry Point Object, then access is allowed.
- Otherwise, if the Java Card RE is the currently active context, then access is allowed.
- Otherwise, access is denied.

6.2.8.9 Accessing Standard Interfaces

Bytecodes:

`checkcast, instanceof`

- If the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the object is designated a Java Card RE Entry Point Object, then access is allowed.
- Otherwise, if the Java Card RE is the currently active context, then access is allowed.
- Otherwise, access is denied.

6.2.8.10 Accessing Shareable Interfaces

Bytecodes:

`checkcast, instanceof`

- If the object is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the object's class implements a `Shareable` interface, and if the object is being cast into (`checkcast`) or is being verified as being an instance of (`instanceof`) an interface that extends the `Shareable` interface, then access is allowed.
- Otherwise, if the Java Card RE is the currently active context, then access is allowed.
- Otherwise, access is denied.

6.2.8.11 Accessing Array Object Methods

Note – The method access behavior of global arrays is identical to that of Java Card RE Entry Point Objects.

Bytecodes:

`invokevirtual`

- If the array is owned by an applet in the currently active context, then access is allowed.
- Otherwise, if the array is designated a global array, then access is allowed. Context is switched to the array owner's context (Java Card RE context).
- Otherwise, if Java Card RE is the currently active context, then access is allowed. Context is switched to the array owner's context.
- Otherwise, access is denied.

CHAPTER 7

Transactions and Atomicity

A *transaction* is a logical set of updates of persistent data. For example, transferring some amount of money from one account to another is a banking transaction. It is important for transactions to be atomic: either all of the data fields are updated, or none are. The Java Card RE provides robust support for atomic transactions, so that card data is restored to its original pre-transaction state if the transaction does not complete normally. This mechanism protects against events such as power loss in the middle of a transaction, and against program errors that might cause data corruption should all steps of a transaction not complete normally.

7.1 Atomicity

Atomicity defines how the card handles the contents of persistent storage after a stop, failure, or fatal exception during an update of a single object or class field or array component. If power is lost during the update, the applet developer shall be able to rely on what the field or array component contains when power is restored.

The Java Card platform guarantees that any update to a single persistent object or class field will be atomic. In addition, the Java Card platform provides single component level atomicity for persistent arrays. That is, if the smart card loses power during the update of a data element (field in an object/class or component of an array) that shall be preserved across CAD sessions, that data element shall be restored to its previous value.

Some methods also guarantee atomicity for block updates of multiple data elements. For example, the atomicity of the `Util.arrayCopy` method guarantees that either all bytes are correctly copied or else the destination array is restored to its previous byte values.

An applet might not require atomicity for array updates. The `Util.arrayCopyNonAtomic` method is provided for this purpose. It does not use the transaction commit buffer even when called with a transaction in progress.

7.2 Transactions

An applet might need to atomically update several different fields or array components in several different objects. Either all updates take place correctly and consistently, or else all fields/components are restored to their previous values.

The Java Card platform supports a transactional model in which an applet can designate the beginning of an atomic set of updates with a call to the `JCSystem.beginTransaction` method. Each object update after this point is conditionally updated. The field or array component appears to be updated—reading the field/array component back yields its latest conditional value—but the update is not yet committed.

When the applet calls `JCSystem.commitTransaction`, all conditional updates are committed to persistent storage. If power is lost or if some other system failure occurs prior to the completion of `JCSystem.commitTransaction`, all conditionally updated fields or array components are restored to their previous values. If the applet encounters an internal problem or decides to cancel the transaction, it can programmatically undo conditional updates by calling `JCSystem.abortTransaction`.

7.3 Transaction Duration

A transaction always ends when the Java Card RE regains programmatic control upon return from the applet's `select`, `deselect`, `process`, `uninstall`, or `install` methods. This is true whether a transaction ends normally, with an applet's call to `commitTransaction`, or with an abortion of the transaction (either programmatically by the applet, or by default by the Java Card RE). For more details on transaction abortion, refer to Section 7.6.

Transaction duration is the life of a transaction between the call to `JCSystem.beginTransaction`, and either a call to `commitTransaction` or an abortion of the transaction.

7.4 Nested Transactions

The model currently assumes that nested transactions are not possible. There can be only one transaction in progress at a time. If `JCSystem.beginTransaction` is called while a transaction is already in progress, then a `TransactionException` is thrown.

The `JCSystem.transactionDepth` method is provided to allow you to determine if a transaction is in progress.

7.5 Tear or Reset Transaction Failure

If power is lost (tear) or the card is reset or some other system failure occurs while a transaction is in progress, then the Java Card RE shall restore to their previous values all fields and array components conditionally updated since the previous call to `JCSystem.beginTransaction`.

This action is performed automatically by the Java Card RE when it reinitializes the card after recovering from the power loss, reset, or failure. The Java Card RE determines which of those objects (if any) were conditionally updated, and restores them.

Note – The contents of an array component which is updated using the `Util.arrayCopyNonAtomic` method or the `Util.arrayFillNonAtomic` method while a transaction is in progress, is not predictable, following a tear or reset during that transaction.

Note – Object space used by instances created during the transaction that failed due to power loss or card reset can be recovered by the Java Card RE.

7.6 Aborting a Transaction

Transactions can be aborted either by an applet or by the Java Card RE.

Note – The contents of an array component which is updated using the `Util.arrayCopyNonAtomic` method or the `Util.arrayFillNonAtomic` method while a transaction is in progress, is not predictable, following the abortion of the transaction.

7.6.1 Programmatic Abortion

If an applet encounters an internal problem or decides to cancel the transaction, it can programmatically undo conditional updates by calling `JCSystem.abortTransaction`. If this method is called, all conditionally updated

fields and array components since the previous call to `JCSystem.beginTransaction` are restored to their previous values, and the `JCSystem.transactionDepth` value is reset to 0.

7.6.2 Abortion by the Java Card RE

If an applet returns from the `select`, `deselect`, `process`, or `install` methods when an applet initiated transaction is in progress, the Java Card RE automatically aborts the transaction and proceeds as if an uncaught exception was thrown.

If the Java Card RE catches an uncaught exception from the `select`, `deselect`, `process`, or `install` methods when an applet initiated transaction is in progress, the Java Card RE automatically aborts the transaction.

Note – The abortion of a transaction by the Java Card RE does not directly affect the response status sent to the CAD. The response status is determined as described in Section 3.3 “The Method process.”

7.6.3 Cleanup Responsibilities of the Java Card RE

Object instances created during the transaction that is being aborted can be deleted only if references to these deleted objects can no longer be used to access these objects. The Java Card RE shall ensure that a reference to an object created during the aborted transaction is equivalent to a `null` reference.

Alternatively, programmatic abortion after creating objects within the transaction can be deemed to be a programming error. When this occurs, the Java Card RE may, to ensure the security of the card and to avoid heap space loss, lock up the card session to force tear/reset processing.

7.7 Transient Objects and Global Arrays

Only updates to persistent objects participate in the transaction. Updates to transient objects and global arrays are never undone, regardless of whether or not they were “inside a transaction.”

7.8 Commit Capacity

Since platform resources are limited, the number of bytes of conditionally updated data that can be accumulated during a transaction is limited. The Java Card technology provides methods to determine how much *commit capacity* is available on the implementation. The commit capacity represents an upper bound on the number of conditional byte updates available. The actual number of conditional byte updates available may be lower due to management overhead.

A `TransactionException` is thrown if the commit capacity is exceeded during a transaction.

7.9 Context Switching

Context switches shall not alter the state of a transaction in progress. If a transaction is in progress at the time of a context switch (see Section 6.1.2), updates to persistent data continue to be conditional in the new context until the transaction is committed or aborted.

CHAPTER 8

Remote Method Invocation Service

Java Card platform Remote Method Invocation ("Java Card RMI") is a subset of the Java Remote Method Invocation (RMI) system. It provides a mechanism for a client application running on the CAD platform to invoke a method on a remote object on the card. The on-card transport layer for Java Card RMI is provided in the package `javacard.framework.service` by the class `RMIService`. It is designed as a service requested by the Java Card RMI-based applet when it is the currently selected applet.

The Java Card RMI message is encapsulated within the APDU object passed into the `RMIService` methods.

8.1 Java Card RMI

This section defines the subset of the RMI system that is supported by Java Card RMI.

8.1.1 Remote Objects

A remote object is one whose remote methods can be invoked remotely from the CAD client. A remote object is described by one or more remote interfaces. A remote interface is an interface which extends, directly or indirectly, the interface `java.rmi.Remote`. The methods of a remote interface are referred to as remote methods. A remote method declaration includes the exception `java.rmi.RemoteException` (or one of its superclasses such as `java.io.IOException` or `java.lang.Exception`) in its `throws` clause. Additionally, in the remote method declaration, a remote object declared as the return value must be declared as the remote interface, not the implementation class of that interface.

Java Card RMI imposes additional constraints on the definition of remote methods. These constraints are a result of the Java Card platform language subset and other feature limitations.

8.1.1.1 Parameters and Return Values

The parameters of a remote method must only include parameters of the following types:

- any supported primitive data types.
- any single-dimension array of a supported primitive data type.

The return value of a remote method must only be one of the following types:

- any supported primitive data type.
- any single-dimension array type of a supported primitive data type.
- any remote interface type
- a void return

All parameters including array parameters are always transmitted by value during the remote method invocation. The return values from a remote method are transmitted by value for primitive types and arrays; return values that are remote object references are transmitted by reference using a remote object reference descriptor.

8.1.1.2 Exceptions

Java Card RMI uses the following simplified model for returning exceptions thrown by remote methods:

- When an exception defined in the Java Card API is thrown by a remote method the exact exception type and the embedded reason code is transmitted back to the client application. In essence, the exception object is transmitted by value.
- When an exception not defined in the Java Card API is thrown by a remote method, the "closest" superclass exception type from the API and the embedded reason code is transmitted back to the client application. In this case, the "closest" API defined superclass exception object is transmitted by value. The client application can distinguish an inexact exception from an exact one.

8.1.1.3 Functional Limitations

The definition of the supported subset of Java Card RMI for Java Card platform version 2.2.1 implies functional limitations during the execution of Java Card API remote methods:

- CAD client application remote objects cannot be passed as arguments to remote methods.

- Card remote objects cannot be passed as arguments to remote methods.
- Applets on the card cannot invoke remote methods on the CAD client.
- Method argument data and return values, along with the Java Card RMI protocol overhead must fit within the size constraints of an APDU command and APDU response respectively.

8.2 The RMI Messages

The Java Card RMI message protocol consists of two commands that are used to:

- get the initial remote object reference for the Java Card RMI based applet. The initial remote object reference is the seed remote object which the CAD client application needs to begin remote method invocations.
- send a remote method invocation request to the card.

To ensure that the protocol is compatible with all applications, the SELECT FILE command is used for getting the initial reference. The response to the SELECT FILE command allows the remote method invocation command itself to be customized by the applet.

8.2.1 Applet Selection

The selection command used to retrieve the initial reference is the ISO7816-4 SELECT FILE command, with the following options in the header:

- Direct selection by DF Name: that is, selection by AID—This is the normal option used to select all applet instances in the Java Card platform.
- Return FCI (File Control Information - ISO7816-4), optional template—This is an additional option that indicates that the applet is expected to return FCI information.

  In addition, an alternate RFU variant of the Return FCI option is required to configure the `RMIService` for an alternate Java Card RMI protocol format. (For more details see Section 8.4.1 “SELECT FILE Command.”)

The answer to this command is a constructed TLV(tag-length-value) data structure (ISO 7816-6) that includes the following information:

- The byte to be used as instruction byte (INS) for subsequent invocation commands.
- The initial remote object reference descriptor. The descriptor includes the remote object identifier and information to identify the associated class.

8.2.2 Method Invocation

In order to request a method invocation, the CAD client provides the following information:

- The remote object identifier—This identifier is used to uniquely identify the object on the card.
- The invoked method identifier—A designator to uniquely identify the remote method within the remote object class or superclass.
- The values of the arguments—These values are raw values for primitive data types, and for arrays, a length followed by the values.

The response to the invocation request may include one of the following items:

- A primitive return value—This is a raw primitive data type value.
- An array of primitive components—This is a length followed by the raw primitive data type values.
- A remote object reference descriptor—The descriptor includes the remote object identifier and information to instantiate a proxy instance of the remote card object.
- An exception thrown by the remote method.

8.3 Data Formats

This section describes the formats used to encapsulate the following:

- A remote object identifier which identifies the remote object on the card.
- A remote object reference descriptor which describes the remote object on the card for the CAD client.
- A method identifier which identifies the remote method on the card.
- The method parameters and return values.

This section uses a C-like structure notation similar to that used in the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*.

8.3.1 Remote Object Identifier

A remote object identifier is a 16-bit unsigned number which uniquely identifies a remote object on the card.

8.3.2 Remote Object Reference Descriptor

The remote object reference descriptor includes the remote object identifier, as well as information to instantiate the proxy class on the CAD client. The remote object reference descriptor uses one of two alternate formats. The representation based on the name of the class uses the `remote_ref_with_class` format. The representation based on the names of the implemented remote interfaces uses the `remote_ref_with_interfaces` format.

A remote object reference descriptor is therefore defined as follows:

```
remote_ref_descriptor {
    union {
        ref_null remote_ref_null
        remote_ref_with_class remote_ref_c
        remote_ref_with_interfaces remote_ref_i
        }
}
```

Note – Even though the above structure uses the C-like "union" notation, the lengths of the alternate representations within the union do not use any padding to normalize their lengths.

The items in the `remote_ref_descriptor` structure are:

`ref_null` is the representation of a null reference using the following format:

```
ref_null {
   u2 remote_ref_id = 0xFFFF
}
```

The `remote_ref_id` item must be the reserved value `0xFFFF`.

`remote_ref_with_class` is the definition of a remote object reference using the class name and uses the following format:

```
remote_ref_with_class {
  u2 remote_ref_id != 0xFFFF
  u1 hash_modifier_length
  u1 hash_modifier[ hash_modifier_length ]
  u1 pkg_name_length
  u1 package_name[ pkg_name_length ]
  u1 class_name_length
  u1 class_name[ class_name_length ]
}
```

The `remote_ref_id` item represents the remote reference identifier. The value of this field must not be `0xFFFF` which denotes the `null` reference.

The `hash_modifier` item is an UTF-8 string of length specified in the `hash_modifier_`length item and is used to ensure that method identifier hash codes are unique.

The pkg_name_length item is the number of bytes in the package_name item to represent the name of the package in UTF-8 string notation. The value of this item must be non-zero.

The package_name item is the variable length representation of the fully qualified name of the package which contains the remote class in UTF-8 string notation. The fully qualified name of the package represented here uses the internal form wherein the ASCII periods ('.') that normally separate the indentifiers that make up the fully qualified name are replaced by ASCII forward slashes ('/'). For example, the internal form of the normally fully qualified package name of the package java.rmi is java/rmi.

The class_name_length item is the number of bytes in the class_name item to represent the name of the remote class in UTF-8 string notation. The value of this item must be non-zero.

The class_name item is the variable length representation of the name of the implementation class of the remote object in UTF-8 string notation.

remote_ref_with_interfaces item is the definition of a remote object reference using the names of the interfaces and uses the following format:

```
remote_ref_with_interfaces {
      u2 remote_ref_id != 0xFFFF
      u1 hash_modifier_length
      u1 hash_modifier[ hash_modifier_length ]
      u1 remote_interface_count
      rem_interface_def remote_interfaces[remote_interface_count]
   }
```

The definition of the remote_ref_id, the hash_modifier_length and the hash_modifier item are the same as that described earlier in the remote_ref_with_class structure.

The remote_interface_count item indicates the number of rem_interface_def format entries in the remote_interfaces item. This number must be less than 16.

The remote_interfaces item comprises a sufficient list of rem_interface_def format entries containing the names of remote interfaces implemented. This list is such that when combined with their remote superinterfaces, the complete set of remote interfaces implemented by the remote object can be enumerated. The rem_interface_def item uses the following format:

```
rem_interface_def {
    u1 pkg_name_length
    u1 package_name[ pkg_name_length ]
    u1 interface_name_length
    u1 interface_name[ interface_name_length ]
    }
```

The items in the rem_interface_def structure are as follows:

The `pkg_name_length` item is the number of bytes used in the `package_name` item to represent the name of the package in UTF-8 string notation. If the value of this item is 0, it indicates that the package name of the previous `remote_interfaces` item must be used instead. The value of this item in `remote_interfaces[0]` must not be `0`.

The `package_name` item is the `pkg_name_length` byte length representation of the fully qualified name of the package which contains the remote interface in UTF-8 string notation. The fully qualified name of the package represented here uses the internal form wherein the ASCII periods (‘`.`’) that normally separate the indentifiers that make up the fully qualified name are replaced by ASCII forward slashes (‘`/`’). For example, the internal form of the normally fully qualified package name of the package `java.rmi` is `java/rmi`.

The `interface_name_length` item is the number of bytes in the `interface_name` item to represent the name of the remote interface in UTF-8 string notation.

The `interface_name` item is the variable length representation of the name of the remote interface implemented by the remote object in UTF-8 string notation.

8.3.3 Method Identifier

A method identifier will always be used in association with a remote object reference. A method identifier is defined as follows:

```
u2 method_id
```

The `method_id` is a unique 16-bit hashcode identifier of the remote method within the remote class. This 16-bit hashcode consists of the first two bytes of the SHA-1 message digest function performed on a class specific hash modifier string followed the name of the method followed by the method descriptor representation in UTF-8 format. Representation of a method descriptor is the same as that described in *The Java Virtual Machine Specification* (Section 4.3.3)

8.3.4 Parameter Encoding

Every parameter will have the following generic format:

```
param {
      u1 value[]
    }
```

8.3.4.1 Primitive Data Type Parameter Encoding

Primitive data types `void`, `boolean`, `byte`, `short` and `int` will respectively be encoded as follows:

```
void_param {
    }
boolean_param {
      u1 boolean_value
    }
byte_param {
      s1 byte_value
    }
short_param {
      s2 short_value
    }
int_param {
      s4 int_value
    }
```

The `boolean_value` field may only take the values `0` (for `false`) and 1 (for `true`). All the other fields can take any value in their range.

8.3.4.2 Array Parameter Encoding

The representation of the null array parameter and arrays of the `boolean`, `byte`, `short` and `int` component types include the length information and will respectively be encoded as follows:

```
null_array_param {
      u1 length = 0xFF
    }
boolean_array_param {
      u1 length != 0xFF
      u1 boolean_value[length]
    }
byte_array_param {
      u1 length != 0xFF
      s1 byte_value[length]
    }
short_array_param {
      u1 length != 0xFF
      s2 short_value[length]
    }
int_array_param {
      u1 length != 0xFF
      s4 int_value[length]
    }
```

Note – The `length` field in each of the array data structure defined above represents the number of elements of the array, and not its size in bytes.

8.3.5 Return Value Encoding

A return value may be any of the parameter types described in the previous section encapsulated within a normal response format. In addition, the return value may represent a remote object reference type, a `null` return type, various exceptions and the error type.

The generic structure of a return value is as follows:

```
return_response {
      u1 tag
      u1[] value
    }
```

The return value using the `return_response` encoding is always followed by a good completion status code of `0x9000` in the response APDU.

8.3.5.1 Normal Response Encoding

A normal response encapsulates primitive return types, arrays of primitive data types using the same format for the `param` item as described in Section 8.3.4 "Parameter Encoding" using the following format:

```
normal_param_response {
      u1 normal_tag = 0x81
      param normal_value
    }
```

The `null_array_param` format described in the Parameter Encoding Section 8.3.4 is not used to represent a `null` array reference. Instead, a null object reference as well as a null array reference share the following common format:

```
normal_null_response {
      u1 normal_tag = 0x81
      ref_null null_array_or_ref
    }
```

In addition, a remote object reference descriptor type is also encapsulated using the normal response format as follows:

```
normal_ref_response {
      u1 normal_tag = 0x81
      remote_ref_descriptor remote_ref
    }
```

8.3.5.2 Exception Response Encoding

The following is the encoding when an API defined exception is thrown by the remote method. It may be returned during any remote method invocation. The `reason` item is the Java Card platform exception reason code, or `0` for a `java.lang`, `java.rmi` or `java.io` exceptions:

```
exception_response {
      u1 exception_tag = 0x82
      u1 exception_type
      s2 reason
    }
```

The values for the `exception_type` item are shown below:

```
java.lang.Throwable = 0x00
java.lang.ArithmeticException = 0x01
java.lang.ArrayIndexOutOfBoundsException = 0x02
java.lang.ArrayStoreException = 0x03
java.lang.ClassCastException = 0x04
java.lang.Exception = 0x05
java.lang.IndexOutOfBoundsException = 0x06
java.lang.NegativeArraySizeException = 0x07
java.lang.NullPointerException = 0x08
java.lang.RuntimeException = 0x09
java.lang.SecurityException = 0x0A
java.io.IOException = 0x0B
java.rmi.RemoteException = 0x0C
javacard.framework.APDUException = 0x20
javacard.framework.CardException = 0x21
javacard.framework.CardRuntimeException = 0x22
javacard.framework.ISOException = 0x23
javacard.framework.PINException = 0x24
javacard.framework.SystemException = 0x25
javacard.framework.TransactionException = 0x26
javacard.framework.UserException = 0x27
javacard.security.CryptoException = 0x30
javacard.framework.service.ServiceException = 0x40
```

The following is the encoding when a user defined exception is thrown by the remote method. The `exception_type` item represents the closest API defined exception type. It may be returned during any remote method invocation. The `reason` item is the Java Card platform exception reason code, or `0` for the subclasses of `java.lang`, `java.rmi` or `java.io` exceptions:

```
exception_subclass_response {
      u1 exception_subclass_tag = 0x83
      u1 exception_type
      s2 reason
    }
```

8.3.5.3 Error Response Encoding

The following encoding represents an error condition on the card. The error may occur due to marshalling, unmarshalling or resource related problems.

```
error_response {
      u1 error_tag = 0x99
      s2 error_detail
```

```
}
```

The values of the `error_detail` item are enumerated below:

- The Remote Object Identifier is invalid or ineligible for Java Card RMI = `0x0001`
- The Remote Method could not be identified = `0x0002`
- `The` Remote Method signature did not match the parameter format = `0x0003`
- Insufficient resources available to unmarshall parameters = `0x0004`
- Insufficient resources available to marshall response = `0x0005`
- Java Card Remote Method Invocation protocol error = `0x0006`
- Internal Error occurred = `0xFFFF`

8.4 APDU Command Formats

Section 8.3 "Data Formats" described the various elements included in the data portion of the Java Card RMI messages. This section describes the complete format of the APDU commands - the header as well as the data portion containing the message elements described earlier.

8.4.1 SELECT FILE Command

The Select command for an RMI based applet must have the following format:

Note – An asterisk (*) indicates binary notation (%) using bit numbering as in ISO7816. Most significant bit = b8. Least significant bit = b1. An x represents a don't care.

TABLE 1 Select File Command

Field	Value	Description
CLA	%b000000xx	The bits (b2,b1*) are used for logical channels
INS	0xA4	SELECT FILE
P1	0x04	Select by AID
P2	%b000x00xx	Return FCI information

The bits (b2,b1*) are used for partial selection if supported. If bit b5* is 1, the remote reference descriptor uses the `remote_ref_with_interfaces` format, otherwise it uses the alternate `remote_ref_with_class` format.

TABLE 2 Remote Reference Descriptor Format

Field	Value	Description
Lc	xx	Length of the AID
Data	*AID*	AID of the applet to be selected (between 5 and 16 bytes)

The format of the response is defined as shown below. Note that the applet may extend the format to include additional information, if necessary before sending the response back to the CAD. The additional information must retain the TLV format and must not introduce any additional information under the `jc_rmi_data_tag`.

```
select_response {
    u1 fci_tag = 0x6F
    u1 fci_length
        u1 application_data_tag = 0x6E
        u1 application_data_length
            u1 jc_rmi_data_tag = 0x5E
            u1 jc_rmi_data_length
            u2 version = 0x0202
            u1 invoke_ins
            union {
                normal_ref_response normal_initial_ref
                normal_null_response null_initial_ref
                error_response initial_ref_error
                } initial_ref
    }
```

The `jc_rmi_data_length` item above is the combined length in bytes of the `version` item, `invoke_ins` item and the `initial_ref` item. The `application_data_length` item is `jc_rmi_data_length + 2`. The `fci_length` item is `application_data_length + 2`.

The response data includes `invoke_ins`, the instruction byte to use in the method invocation command. It also includes `initial_ref`, the initial remote object reference descriptor. The `initial_ref` item corresponds to the remote object designated as the initial reference to the `RMIService` instance during construction. The `initial_ref` item can be a `normal_ref_response` item described in Section 8.3.5.1 or a `null` representation using a `normal_null_response` item described in Section 8.3.5.1, if the initial remote reference object is not enabled for remote access. Also, note that if an error occurs during the marshalling of the initial remote reference descriptor, an error response is returned in `initial_ref` instead using the `error_response` item format described in Section 8.3.5.3 .

Note – Even though the `select_response` structure shown above uses the C-like "union" notation, the lengths of the alternate representations within the union do not use any padding to normalize their lengths.

The format of the `remote_ref_descriptor` to be used in this response as well as all subsequent responses - `remote_ref_with_class` or `remote_ref_with_interfaces` - is determined by the value of the P2 byte of the SELECT FILE command.

Note – Only the `RMIService` instance which processes the SELECT FILE command sets (or changes) the format of the remote object reference descriptor based on the value of the P2 byte. Once set or changed, the `RMIService` instance uses only that format in all Java Card RMI responses it generates.

8.4.2 INVOKE Command

The Invoke command for a remote method invocation request must have the following format:

TABLE 3 Invoke Command Format

Field	Value	Description
CLA	%b1000xxxx	(b4,b3*) for secure messaging (ISO 7816-4) and (b2,b1*) for logical channels
INS	value of `invoke_ins`	`invoke_ins` returned in the previous `select_response`
P1	02	RMI major version #
P2	02	RMI minor version #
Data	As described below	As described below

The data part of the request command is structured as:

```
invoke_data {
     u2 object_id
     u2 method_id
     param parameters[]
   }
```

The `object_id` is the remote object identifier of the object whose remote method is to be invoked. The method to be invoked is specified by the `method_id` item, and each `parameter` is specified by a `param` structure.

The response format uses the `return_response` structure as described in Section 8.3.5 "Return Value Encoding" above.

8.5 The RMIService Class

The `RMIService` class implements the Java Card RMI protocol and processes the RMI access commands described earlier: SELECT FILE and INVOKE. It performs the function of the transport layer for Java Card RMI commands on the card.

The `RMIService` object maintains a list of remote objects which have been returned during the current applet selection session. It enforces the following rules for the lifetime of the remote object references:

- A remote reference is valid only when the INVOKE command is processed by the `RMIService` instance which returned the reference.
- A remote reference is valid with any applet instance in the package of the applet instance which returned it.
- A remote reference is valid as long as at least one applet instance within the same package has been active at all times since the point in time when the remote reference was returned.
- A valid remote object cannot be garbage collected during the lifetime of the remote reference.

In addition, a remote object reference descriptor of an object must only be returned from the card if it is exported (See the class `javacard.framework.service.CardRemoteObject`.) Otherwise, an exception is thrown (See the class `javacard.framework.service.RMIService`.)

8.5.1 setInvokeInstructionByte Method

This method sets the value of `invoke_ins` described in Section 8.4.1, which is returned in the response to the SELECT FILE command. The change in the Java Card RMI protocol only goes into effect the next time this `RMIService` instance processes the SELECT FILE command. If this method is not called, the default instruction byte value (`DEFAULT_RMI_INVOKE_INSTRUCTION`) is used.

8.5.2 processCommand Method

The `processCommand` method of the `RMIService` class is invoked by the applet to process an incoming RMI message. `RMIService` collaborates with other services by using the common service format (CSF) in the APDU buffer. It processes only the incoming Java Card RMI APDU commands and produces output as described in the previous sections.

When called with a SELECT FILE command with format described in Section 8.4.1, this method builds a response APDU as described in that section.

When called with an INVOKE command with the format described in Section 8.4.2, this method must call the specified remote method of the identified remote object with the specified parameters. It must catch all exceptions thrown by the remote method. When an exception is caught or the remote method returns, this method must build a response APDU in the format described in Section 8.4.2.

Prior to invoking the remote method the following errors must be detected and must result in an error response in the format described in Section 8.3.5.3:

- The remote object identifier is not valid.
- The remote object identifier has not been returned during the current selection session.
- The method identifier does not match any remote methods in the remote class associated with the identified remote object.
- The length of the INVOKE message is inconsistent with the signature of the remote method.
- There is insufficient space to allocate array parameters for the remote method. The implementation must support at least 8 input parameters of type array.

In addition, upon return from the remote method, the following errors must be detected and must result in an error response in the format described in Section 8.3.5.3:

- There is insufficient space to allocate the array response from the remote method. The implementation must support an APDU buffer of at least 133 bytes.
- A remote object is being returned, and its associated remote object identifier has not been previously returned during the current selection session and there is insufficient space to add the remote object identifier to the session remote object identifier list. The implementation must support at least 8 remote object identifiers during a selection session.

In addition to the above, the object access firewall rules must be enforced in a manner similar to that of the `invokevirtual` instruction (Section 6.2.8.4) by this method when a remote method is invoked. Only methods of a remote object owned by the context of the currently selected applet may be invoked.

Allocation of Incoming Objects

Since array parameters to remote methods are transmitted by value, array objects need to be allocated on the card when a remote method with array arguments is invoked via the INVOKE command. Global array objects (Section 6.2.2) must be used for incoming remote method arguments. Global arrays have the following properties:

- They are owned by the Java Card RE, but they can be freely accessed from all contexts.
- They are temporary objects and cannot be stored in any object.
- They are not subject to transactions

The implementation may choose to maintain the data portion of these global array objects used for remote method parameters in the APDU buffer itself.

CHAPTER 9

API Topics

The topics in this chapter complement the requirements specified in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1*.

9.1 Resource Use within the API

Unless specified in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1*, the implementation shall support the invocation of API instance methods, even when the owner of the object instance is not the currently selected applet. In other words, unless specifically called out, the implementation shall not use resources such as transient objects of `CLEAR_ON_DESELECT` type.

9.2 Exceptions thrown by API Classes

All exception objects thrown by the API implementation shall be temporary Java Card RE Entry Point Objects. Temporary Java Card RE Entry Point Objects cannot be stored in class variables, instance variables or array components (See Section 6.2.1).

9.3 Transactions within the API

Unless explicitly called out in the API descriptions, implementation of the Java Card API methods shall not initiate or otherwise alter the state of a transaction in progress.

Unless explicitly called out in the API descriptions, updates to internal implementation state within the API objects must be conditional. In other words, internal state updates must participate in any ongoing transaction.

9.4 The APDU Class

The `APDU` class encapsulates access to the ISO 7816-4 based I/O across the card serial line. The `APDU` class is designed to be independent of the underlying I/O transport protocol.

The Java Card RE may support T=0 or T=1 transport protocols or both.

9.4.1 T=0 Specifics for Outgoing Data Transfers

The `setOutgoing` and `setOutgoingNoChaining` methods in the `APDU` class are used to specify that data needs to be returned to the CAD. These methods return the expected length (`Le`) value as follows:

`ISO 7816-4 CASE 1:` Not applicable. Assume Case 2

`ISO 7816-4 CASE 2: P3` (If `P3=0, 256`)

`ISO 7816-4 CASE 3:` Not applicable. Assume Case 4

`ISO 7816-4 CASE 4: 256`

For compatibility with legacy CAD/terminals that do not support block chained mechanisms, the `APDU` class allows a non-chained transfer mode selection via the `setOutgoingNoChaining` method. The related behaviors are discussed in the following sections.

9.4.1.1 Constrained Transfers with No Chaining

When the no chaining mode of output transfer is requested by the applet by calling the `setOutgoingNoChaining` method, the following protocol sequence shall be followed:

When the no chaining mode is used (that is, after the invocation of the `setOutgoingNoChaining` method), calls to the `waitExtension` method shall throw an `APDUException` with reason code `ILLEGAL_USE`.

Notation

Le = CAD expected length.

Lr = Applet response length set via `setOutgoingLength` method.

<INS> = the protocol byte equal to the incoming header INS byte, which indicates that all data bytes will be transferred next.

<~INS> = the protocol byte that is the complement of the incoming header INS byte, which indicates that 1 data byte will be transferred next.

<SW1,SW2> = the response status bytes as in ISO7816-4.

ISO 7816-4 CASE 2

Le == Lr

1. The card sends Lr bytes of output data using the standard T=0 <INS> or <~INS> procedure byte mechanism.
2. The card sends <SW1,SW2> completion status on completion of the `Applet.process` method.

Lr < Le

1. The card sends <0x61,Lr> completion status bytes
2. The CAD sends GET RESPONSE command with Le = Lr.
3. The card sends Lr bytes of output data using the standard T=0 <INS> or <~INS> procedure byte mechanism.
4. The card sends <SW1,SW2> completion status on completion of the `Applet.process` method.

Lr > Le

1. The card sends Le bytes of output data using the standard T=0 <INS> or <~INS> procedure byte mechanism.
2. The card sends <0x61,(Lr-Le)> completion status bytes
3. The CAD sends GET RESPONSE command with new Le <= Lr.
4. The card sends (new) Le bytes of output data using the standard T=0 <INS> or <~INS> procedure byte mechanism.
5. Repeat steps 2-4 as necessary to send the remaining output data bytes (Lr) as required.
6. The card sends <SW1,SW2> completion status on completion of the `Applet.process` method.

ISO 7816-4 CASE 4

In Case 4, Le is determined after the following initial exchange:

1. The card sends <0x61,Lr status bytes>
2. The CAD sends GET RESPONSE command with Le <= Lr.

The rest of the protocol sequence is identical to CASE 2 described above.

If the applet aborts early and sends less than Le bytes, zeros shall be sent instead to fill out the length of the transfer expected by the CAD.

9.4.1.2 Regular Output Transfers

When the no chaining mode of output transfer is not requested by the applet (that is, the `setOutgoing` method is used), any ISO-7816-3/4 compliant T=0 protocol transfer sequence may be used.

Note – The waitExtension method may be invoked by the applet at any time. The waitExtension method shall request an additional work waiting time (ISO 7816-3) using the 0x60 procedure byte.

9.4.1.3 Additional T=0 Requirements

At any time, when the T=0 output transfer protocol is in use, and the APDU class is awaiting a GET RESPONSE command from the CAD in reaction to a response status of <0x61, xx> from the card, if the CAD sends in a different command on the same origin logical channel, or a command on a different origin logical channel, the sendBytes or the sendBytesLong methods shall throw an `APDUException` with reason code `NO_T0_GETRESPONSE`.

At any time, when the T=0 output transfer protocol is in use, and the APDU class is awaiting a command reissue from the CAD in reaction to a response status of <0x6C, xx> from the card, if the CAD sends in a different command on the same origin logical channel, or a command on a different origin logical channel, the `sendBytes` or the `sendBytesLong` methods shall throw an `APDUException` with reason code `NO_T0_REISSUE`.

Calls to `sendBytes` or `sendBytesLong` methods after the `NO_T0_GETRESPONSE` exception or the `NO_T0_REISSUE` exception has been thrown, shall result in an `APDUException` with reason code `ILLEGAL_USE`. If an `ISOException` is thrown by the applet after the `NO_T0_GETRESPONSE` exception or the `NO_T0_REISSUE` exception has been thrown, the Java Card RE shall discard the response status in its reason code. The Java Card RE shall restart APDU processing with the newly received command and resume APDU dispatching.

9.4.2 T=1 Specifics for Outgoing Data Transfers

The `setOutgoing` and `setOutgoingNoChaining` methods in the `APDU` class are used to specify that data needs to be returned to the CAD. These methods return the expected length (`Le`) value as follows:

```
ISO 7816-4 CASE 1: 0
ISO 7816-4 CASE 2: Le
ISO 7816-4 CASE 3: 0
ISO 7816-4 CASE 4: Le
```

9.4.2.1 Constrained Transfers with No Chaining

When the no chaining mode of output transfer is requested by the applet by calling the `setOutgoingNoChaining` method, the following protocol specifics shall be followed:

Notation

Le = CAD expected length.

Lr = Applet response length set via `setOutgoingLength` method.

The transport protocol sequence shall not use block chaining. Specifically, the M-bit (more data bit) shall not be set in the PCB of the I-blocks during the transfers (ISO 7816-3). In other words, the entire outgoing data (Lr bytes) shall be transferred in one I-block.

If the applet aborts early and sends less than Lr bytes, zeros shall be sent instead to fill out the remaining length of the block.

Note – When the no chaining mode is used (i.e. after the invocation of the `setOutgoingNoChaining` method), calls to the `waitExtension` method shall throw an `APDUException` with reason code `ILLEGAL_USE`.

9.4.2.2 Regular Output Transfers

When the no chaining mode of output transfer is not requested by the applet (i.e. the `setOutgoing` method is used) any ISO-7816-3/4 compliant T=1 protocol transfer sequence may be used.

Note – The `waitExtension` method may be invoked by the applet at any time. The `waitExtension` method shall send an S-block command with WTX request of INF units, which is equivalent to a request of 1 additional work waiting time in T=0 mode. (See ISO 7816-3).

Chain Abortion by the CAD

If the CAD aborts a chained outbound transfer using an S-block ABORT request (see ISO 7816-3), the `sendBytes` or `sendBytesLong` method shall throw an `APDUException` with reason code `T1_IFD_ABORT`.

Calls to `sendBytes` or `sendBytesLong` methods from this point on shall result in an APDUException with reason code `ILLEGAL_USE`. If an `ISOException` is thrown by the applet after the `T1_IFD_ABORT` exception has been thrown, the Java Card RE shall discard the response status in its reason code. The Java Card RE shall restart APDU processing with the newly received command, and resume APDU dispatching.

9.4.3 T=1 Specifics for Incoming Data Transfers

9.4.3.1 Incoming Transfers using Chaining

Chain Abortion by the CAD

If the CAD aborts a chained inbound transfer using an S-block ABORT request (see ISO 7816-3), the `setIncomingAndReceive` or `receiveBytes` method shall throw an `APDUException` with reason code `T1_IFD_ABORT`.

Calls to `receiveBytes`, `sendBytes` or `sendBytesLong` methods from this point on shall result in an `APDUException` with reason code `ILLEGAL_USE`. If an `ISOException` is thrown by the applet after the `T1_IFD_ABORT` exception has been thrown, the Java Card RE shall discard the response status in its reason code. The Java Card RE shall restart APDU processing with the newly received command, and resume APDU dispatching.

9.5 The Security and Crypto Packages

The `getInstance` method in the following classes return an implementation instance in the context of the calling applet of the requested algorithm:

```
javacard.security.MessageDigest
javacard.security.Signature
javacard.security.RandomData
javacard.security.KeyAgreement
javacard.security.Checksum
javacardx.crypto.Cipher
```

An implementation of the Java Card RE may implement 0 or more of the algorithms listed in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1*. When an algorithm that is not implemented is requested this method shall throw a `CryptoException` with reason code `NO_SUCH_ALGORITHM`.

Implementations of the above classes shall extend the corresponding base class and implement all the abstract methods. All data allocation associated with the implementation instance shall be performed at the time of instance construction to ensure that any lack of required resources can be flagged early during the installation of the applet.

Similarly, the `buildKey` method of the `javacard.security.KeyBuilder` class returns an implementation instance of the requested Key type. The Java Card RE may implement 0 or more types of keys. When a key type that is not implemented is requested, the method shall throw a `CryptoException` with reason code `NO_SUCH_ALGORITHM`.

In the same fashion, the constructor for the `javacard.security.KeyPair` class creates a `KeyPair` instance for the specified key type. The Java Card RE may implement 0 or more types of keys. When a key type that is not implemented is requested, the method shall throw a `CryptoException` with reason code `NO_SUCH_ALGORITHM`.

Implementations of key types shall implement the associated interface. All data allocation associated with the key implementation instance shall be performed at the time of instance construction to ensure that any lack of required resources can be flagged early during the installation of the applet.

The `MessageDigest` object uses temporary storage for intermediate results when the `update()` method is invoked. This intermediate state need not be preserved across power up and reset. The object is reset to the state it was in when previously initialized via a call to `reset()`.

The `Signature` and `Cipher` objects use temporary storage for intermediate results when the `update()` method is invoked. This intermediate state need not be preserved across power up and reset. The object is reset to the state it was in when previously initialized via a call to `init()`.

The Checksum object uses temporary storage for intermediate results when the `update()` method is invoked. This intermediate state need not be preserved across power up and reset. The object is reset to the state it was in when previously initialized upon a tear or card reset event.

9.6 JCSystem Class

In the Java Card platform, version 2.2.1, the `getVersion` method returns (`short`) 0x0202.

CHAPTER 10

Virtual Machine Topics

The topics in this chapter detail virtual machine specifics.

10.1 Resource Failures

A lack of resources condition (such as heap space) which is recoverable shall result in a `SystemException` with reason code `NO_RESOURCE`. The factory methods in `JCSystem` used to create transient arrays throw a `SystemException` with reason code `NO_TRANSIENT_SPACE` to indicate lack of transient space.

All other (non-recoverable) virtual machine errors such as stack overflow shall result in a virtual machine error. These conditions shall cause the virtual machine to halt. When such a non-recoverable virtual machine error occurs, an implementation can optionally require the card to be muted or blocked from further use.

10.2 Security Violations

The Java Card RE throws a `java.lang.SecurityException` exception when it detects an attempt to illegally access an object belonging to another applet across the firewall boundary. A `java.lang.SecurityException` exception may optionally be thrown by a Java Card VM implementation to indicate a violation of fundamental language restrictions, such as attempting to invoke a private method in another class.

For security reasons, the Java Card RE implementation may mute the card instead of throwing the exception object.

CHAPTER 11

Applet Installation and Deletion

Applet installation and deletion on smart cards using Java Card technology is a complex topic. The design of the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* is intended to give Java Card RE implementers as much freedom as possible in their implementations. However, some basic common specifications are required in order to allow Java Card applets to be installed and deleted without knowing the implementation details of a particular installer or deletion manager.

This specification defines the concepts of an Installer and an Applet Deletion Manager, and specifies minimal requirements in order to achieve interoperability across a wide range of possible Installer implementations.

The Applet Installer is an optional part of the *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1*. That is, an implementation of the Java Card RE does not necessarily need to include a post-issuance Installer. However, if implemented, the installer is required to support the behavior specified in this chapter.

If the implementation of the Java Card RE includes a post-issuance Installer, then an Applet Deletion Manager which supports the behavior specified in this chapter is also required.

The sections - Section 11.1 "The Installer" and Section 11.2 "The Newly Installed Applet" describe CAP file[1] loading and linking, and applet installation respectively. Even though the loading, and linking operations are described together with the installation operations, there is no requirement that they be performed together during the same card session:

- Applet packages in ROM are pre-loaded and pre-linked at card issuance, but instances of applets from these packages may be installed by the Installer during during a card session.
- Applet packages may be downloaded and linked by the Installer during one card session, but applet instances from these packages may be installed by the Installer during a different card session.

1. See *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1.*

- Library packages may be pre-loaded in ROM or downloaded and linked by the Installer during a card session. There are no applets to install within a library package.

11.1 The Installer

The mechanisms necessary to install an applet on smart cards using Java Card technology are embodied in an on-card component called the *Installer*.

To the CAD the Installer appears to be an applet. It has an AID, and it becomes the currently selected applet when this AID is successfully processed by a SELECT FILE command. Once selected on a logical channel, the Installer behaves in much the same way as any other applet:

- It receives all APDUs dispatched to this logical channel just like any other active applet.
- Its design specification prescribes the various kinds and formats of APDUs that it expects to receive along with the semantics of those commands under various preconditions.
- It processes and responds to all APDUs that it receives. Incorrect APDUs are responded to with an error condition of some kind.
- When another applet is selected on this logical channel(or when the card is reset or when power is removed from the card), the Installer becomes deselected and remains suspended until the next time that it is selected.

11.1.1 Installer Implementation

The Installer need not be implemented as an applet on the card. The requirement is only that the Installer functionality be SELECTable. The corollary to this requirement is that Installer component shall not be able to be invoked on a logical channel on which a non-Installer applet is an active applet instance nor when no applet is active.

Obviously, a Java Card RE implementer could choose to implement the Installer as an applet. If so, then the Installer might be coded to extend the `Applet` class and respond to invocations of the `select`, `process`, and `deselect` methods; and, if necessary, the methods of the `javacard.framework.MultiSelectable` interface.

But a Java Card RE implementer could also implement the Installer in other ways, as long as it provides the SELECTable behavior to the outside world. In this case, the Java Card RE implementer has the freedom to provide some other mechanism by which APDUs are delivered to the Installer code module.

11.1.2 Installer AID

Because the Installer is SELECTable, it shall have an AID. Java Card RE implementers are free to choose their own AIDs by which their Installer is selected. Multiple installers may be implemented.

11.1.3 Installer APDUs

The Java Card specification does not specify any APDUs for the Installer. Java Card RE implementers are entirely free to choose their own APDU commands to direct their Installer in its work.

The model is that the Installer on the card is initiated by an installation program running on the CAD. In order for installation to succeed, this CAD installation program shall be able to:

- Recognize the card.
- SELECT FILE the Installer on the card.
- Coordinate the installation process by sending the appropriate APDUs to the card Installer. These APDUs will include:
  - Authentication information, to ensure that the installation is authorized.
  - The applet code to be loaded into the card's memory.
  - Linkage information to link the applet code with code already on the card.
  - Instance initialization parameter data to be sent to the applet's `install` method.

The *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* does not specify the details of the CAD installation program nor the APDUs passed between it and the Installer.

11.1.4 CAP File Versions

The Installer shall support the following CAP file versions:

- Version 2.1 as specified in the *Java Card™ 2.1.1 Virtual Machine Specification.*
- Version 2.2 as specified in the *Java Card™ 2.2 Virtual Machine Specification.*

11.1.5 Installer Behavior

Java Card RE implementers shall also define other behaviors of their Installer, including:

- Whether or not installation can be aborted and how this is done.
- What happens if an exception, reset, or power fail occurs during installation.

- What happens if another applet is selected before the Installer is finished with its work.

The Java Card RE shall guarantee that an applet will *not* be deemed successfully installed if:

- the applet package as identified by the package AID is already resident on the card.
- the applet package contains an applet with the same Java Card platform name as that of another applet already resident on the card. The Java Card platform name of an applet identified by the AID item is described in section 6.5 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1*.
- the applet package requires more memory than is available on the card.
- the applet package references a package that is not resident on the card.
- the applet package references another package already resident on the card, but the version of the resident package is not binary compatible with the applet package. For more information on binary compatibility in the Java programming language please see the *Java™ Language Specification.* Binary compatibility in Java Card technology is discussed in the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1.*
- a class in the applet package is found to contain more package visible virtual methods or instance fields than the limitations enumerated in section 2.2.4.3 of the *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1.*
- a reset or power fail occurs while executing the applet's `install` method and before successful return from the `Applet.register` method (see Section 3.1).
- the applet's `install` method throws an exception before successful return from the `Applet.register` method (see Section 3.1).

When applet installation is unsuccessful, the Java Card RE shall guarantee that objects created during the execution of the `install` method, or by the Java Card RE on its behalf—initialized static arrays—can never be accessed by any applet on the card. In particular, any reference in CLEAR_ON_RESET transient space to an object created during an unsuccessful applet installation must be reset as a `null` reference.

11.1.6 Installer Privileges

Although an Installer may be implemented as an applet, an Installer will typically require access to features that are not available to "other" applets. For example, depending on the Java Card RE implementer's implementation, the Installer will need to:

- Read and write directly to memory, bypassing the object system and/or standard security.
- Access objects owned by other applets or by the Java Card RE.
- Invoke non-entry point methods of the Java Card RE.

- Be able to invoke the install method of a newly installed applet.

Again, it is up to each Java Card RE implementer to determine the Installer implementation and supply such features in their Java Card RE implementations as necessary to support their Installer. Java Card RE implementers are also responsible for the security of such features, so that they are not available to normal applets.

11.2 The Newly Installed Applet

There is a single interface between the Installer and the applet that is being installed. After the Installer has correctly prepared the applet for execution (performed steps such as loading and linking), the Installer shall invoke the applet's `install` method. This method is defined in the `Applet` class.

The precise mechanism by which an applet's `install(byte[], short, byte)` method is invoked from the Installer is a Java Card RE implementer-defined implementation detail. However, there shall be a context switch so that any context-related operations performed by the `install` method (such as creating new objects) are done in the context of the new applet and not in the context of the Installer. The Installer shall also ensure that array objects created in the class initialization (<clinit>) methods of the applet package are also owned by the context of the new applet.

The Installer shall not invoke the `install(byte[], short, byte)` method of an applet if another applet from the same package is active on the card. The applet instantiation shall be deemed unsuccessful.

The Installer shall ensure that during the execution of the `install()` method, the new applet (not the Installer) is the *currently selected applet*. In addition, any `CLEAR_ON_DESELECT` objects created during the `install()` method shall be associated with the selection context of the new applet.

The installation of an applet is deemed complete if all steps are completed without failure or an exception being thrown, up to and including successful return from executing the `Applet.register` method. At that point, the installed applet will be selectable.

The maximum size of the parameter data is 127 bytes. The `bArray` parameter is a global array (`install(byte[] bArray, short bOffset, byte bLength)`), and for security reasons is zeroed after the return from the `install` method, just as the APDU buffer is zeroed on return from an applet's process method.

11.2.1 Installation Parameters

The format of the input data passed to the target applet's install method in the `bArray` parameter is as follows:

```
bArray[offset] = length(Li) of instance AID
bArray[offset+1..offset+Li] = instance AID bytes (5-16 bytes)
bArray[offset+Li+1]= length(Lc) of control info
bArray[offset+Li+2..offset+Li+Lc+1] = control info
bArray[offset+Li+Lc+2] = length(La) of applet data
bArray[offset+Li+Lc+2..offset+Li+Lc+La+1] = applet data
```

Any of the length items: `Li, Lc, La` may be zero. If length `Li` is non-zero, the `instance AID bytes` item is the proposed AID of the applet instance.

The `control info` item of the parameter data is implementation dependent and is specified by the Installer.

Other than the need for the entire parameter data to not be greater than 127 bytes, the Java Card API does not specify anything about the contents of the `applet data` item of the global byte array installation parameter. This is fully defined by the applet designer and can be in any format desired. In addition, the `applet data` portion is intended to be opaque to the Installer.

Java Card RE implementers should design their Installers so that it is possible for an installation program running in a CAD to specify the `applet data` delivered to the Installer. The Installer simply forwards this along with the other items in the format defined above to the target applet's `install` method in the `bArray` parameter. A typical implementation might define a Java Card RE implementer-proprietary APDU command that has the semantics "call the applet's `install` method passing the contents of the accompanying `applet data`."

11.3 The Applet Deletion Manager

The mechanisms necessary to delete an applet on smart cards using Java Card technology are embodied in an on-card component called the *Applet Deletion Manager*.

To the CAD, the Applet Deletion Manager appears to be an applet, and may be one and the same as the Applet Installer. It has an AID, and it becomes the currently selected applet instance when this AID is successfully processed by a SELECT FILE command. Once selected on a logical channel, the Applet Deletion Manager behaves in much the same way as any other applet:

- It receives all APDUs dispatched to this logical channel, just like any other active applet.
- Its design specification prescribes the various kinds and formats of APDUs that it expects to receive, along with the semantics of those commands under various preconditions.
- It processes and responds to all APDUs that it receives. Incorrect APDUs are responded to with an error condition of some kind.

- When another applet is selected on this logical channel (or when the card is reset or when power is removed from the card), the Applet Deletion Manager becomes deselected and remains suspended until the next time that it is selected.

11.3.1 Applet Deletion Manager Implementation

The Applet Deletion Manager need not be implemented as an applet on the card. The requirement is only that the Applet Deletion Manager functionality be SELECTable. The corollary to this requirement is that Applet Deletion Manager component shall not be able to be invoked on a logical channel where a non-Applet Deletion Manager applet is an active applet instance, nor when no applet is active.

Obviously, a Java Card RE implementer could choose to implement the Applet Deletion Manager as an applet. If so, then the Applet Deletion Manager might be coded to extend the `Applet` class and to respond to invocations of the `select`, `process`, and `deselect` methods, and if necessary the methods of the `javacard.framework.MultiSelectable` interface.

But a Java Card RE implementer could also implement the Applet Deletion Manager in other ways, as long as it provides the SELECTable behavior to the outside world. In this case, the Java Card RE implementer has the freedom to provide some other mechanism by which APDUs are delivered to the Applet Deletion Manager code module.

11.3.2 Applet Deletion Manager AID

Because the Applet Deletion Manager is SELECTable, it shall have an AID which may be the same as that of the Applet Installer. Java Card RE implementers are free to choose their own AIDs by which their Applet Deletion Manager is selected. Multiple Applet Deletion Managers may be implemented.

11.3.3 Applet Deletion Manager APDUs

The Java Card API does not specify any APDUs for the Applet Deletion Manager. Java Card RE implementers are entirely free to choose their own APDU commands to direct their Applet Deletion Manager in its work.

The model is that the Applet Deletion Manager on the card is initiated by an applet deletion program running on the CAD. In order for applet deletion to succeed, this CAD applet deletion program shall be able to:

- Recognize the card.
- SELECT FILE the Applet Deletion Manager on the card.
- Coordinate the applet deletion process by sending the appropriate APDUs to the card Applet Deletion Manager. These APDUs will include:

- Authentication information, to ensure that the applet deletion is authorized.
- Identify the applet(s) code or instance to be deleted from the card's memory.

The *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* does not specify the details of the CAD applet deletion program nor the APDUs passed between it and the Applet Deletion Manager.

11.3.4 Applet Deletion Manager Behavior

Java Card RE implementers shall also define other behaviors of their Applet Deletion Manager, including:

- Whether or not applet deletion can be aborted and how this is done.
- What happens if an exception, reset, or power fail occurs during applet deletion.
- What happens if another applet is selected before the Applet Deletion Manager is finished with its work.

There are three categories of applet deletion requirements on the card:

- Applet instance deletion involves the removal of the applet object instance and the objects owned by the applet instance and associated Java Card RE structures.
- Applet/library package deletion involves the removal of all the card resident components of the CAP file, including code and any associated Java Card RE management structures.
- Deletion of the applet package and the contained applet instances involves the removal of the card-resident code and Java Card RE structures associated with the applet package, and all the applet instances and objects in the context of the package and associated Java Card RE structures.

Invocation of the `javacard.framework.AppletEvent.uninstall` method

Whenever one or more applet instances is being deleted, the Applet Deletion Manager shall inform each of the applets of potential deletion by invoking, if implemented, the applet's `uninstall` method. When multiple applet instances are being deleted, the order of invocation of the uninstall methods is unspecified. Prior to following the stepwise sequence described in Section 11.3.4.1, Section 11.3.4.2, or Section 11.3.4.3, the Java Card RE shall:

- perform any security and authorization checks required for the deletion of each of the applet instances to be deleted. If the checks fail, an error is returned and the applet deletion fails.
- otherwise, check if an applet instance, belonging to the contexts of the applet instances being deleted, is active on the card. If so, an error is returned and the applet instance deletion fails.
- otherwise, perform the following steps for each of the applet instances to be deleted :

- if the applet instance being deleted implements the `AppletEvent` interface, set the currently selected applet to that of the applet instance and invoke the `uninstall` method of the applet instance.
  - A context switch into the context of the applet instance occurs upon invocation.
  - if an uncaught exception is thrown during the execution of the `uninstall` method, it is caught and ignored.

11.3.4.1 Applet Instance Deletion

The Java Card RE shall guarantee that applet instance deletion will not be attempted and thereby deemed unsuccessful if:

- any object owned by the applet instance is referenced from an object owned by another applet instance on the card, or
- any object owned by the applet instance is referenced from a static field on any package on the card, or
- an applet instance, belonging to the context of the applet instance being deleted, is active on the card.

Otherwise, the Java Card RE shall delete the applet instance.

Note – The applet deletion attempt may fail due to security considerations or resource limitations.

The applet instance deletion operation must be atomic. If a reset or power fail occurs during the deletion process, it must result in either an unsuccessful applet instance deletion or a successfully completed applet instance deletion before any applet is selected on the card.

Following an unsuccessful applet instance deletion, the applet instance shall be selectable, and all objects owned by the applet shall remain unchanged. In other words, the functionality of all applet instances on the card remains the same as prior to the unsuccessful attempt.

Following a successful applet instance deletion, it shall not be possible to select that applet, and no object owned by the applet can be accessed by any applet currently on the card or by a new applet created in the future.

The resources used by the applet instance may be recovered for re-use.

The AID of the deleted applet instance may be re-assigned to a new applet instance.

Multiple Applet Instance Deletion

The Java Card RE shall guarantee that multiple applet instance deletion will not be attempted, and thereby deemed unsuccessful if:

- an object owned by any of the applet instances being deleted is referenced from an object owned by an applet instance on the card which is not being deleted, or
- an object owned by any of the applet instances being deleted is referenced from a static field on any package on the card, or
- an applet instance, belonging to the contexts of any of the applet instances being deleted is active on the card.

Otherwise, the Java Card RE shall delete the applet instances.

Note – The applet deletion attempt may fail due to security considerations or resource limitations.

The multiple applet instance deletion operation must be atomic. If a reset or power fail occurs during the deletion process, it must result in either an unsuccessful multiple applet instance deletion or a successfully completed multiple applet instance deletion before any applet is selected on the card.

Following an unsuccessful multiple applet instance deletion, all applet instances shall be selectable, and all objects owned by the applets shall remain unchanged. In other words, the functionality of all applet instances on the card remains the same as prior to the unsuccessful attempt.

Following a successful multiple applet instance deletion, it shall not be possible to select any of the deleted applets, and no object owned by the deleted applets can be accessed by any applet currently on the card or by a new applet created in the future.

The resources used by the applet instances may be recovered for re-use.

The AID of the deleted applet instances may be re-assigned to new applet instances.

11.3.4.2 Applet/Library Package deletion

The Java Card RE shall guarantee that applet/library package deletion will not be attempted and thereby deemed unsuccessful if:

- a reachable (non-garbage) instance of a class belonging to the package being deleted exists on the card, or
- another package on the card depends on this package (as expressed in the CAP file's import component).

Otherwise, if the applet/library package is resident in mutable memory, then the Java Card RE shall delete the applet/library package.

Note – The package deletion attempt may fail due to security considerations or resource limitations.

The applet/library package deletion operation must be atomic. If a reset or power fail occurs during the deletion process, it must result in either an unsuccessful applet/library package deletion or a successfully completed applet/library package deletion before any applet is selected on the card.

Following an unsuccessful applet/library package deletion, any object or package that depends on the package continues to function unaffected. In other words, the functionality of all applets on the card remains the same as prior to the unsuccessful attempt.

Following a successful applet/library package deletion, it shall not be possible to install another package which depends on the deleted package. Additionally, it shall be possible to re-install the same package (with exactly the same package AID) or an upgraded version of the deleted package onto the card.

The resources used by the applet/library package may be recovered for re-use.

11.3.4.3 Deletion of Applet Package and Contained Instances

The Java Card RE shall guarantee that deletion of the applet package and contained instances will not be attempted and thereby deemed unsuccessful if:

- another package on the card depends on this package (as expressed in the CAP file's import component), or
- an object owned by any of the applet instances being deleted is referenced from an object owned by an applet instance on the card which is not being deleted, or
- an object owned by any of the applet instances being deleted is referenced from a static field of a package which is not being deleted, or
- an applet instance, belonging to the contexts of any of the applet instances being deleted, is active on the card.

Otherwise, if the applet package is resident in mutable memory, then the Java Card RE shall delete the applet package and contained instances.

Note – The applet and package deletion attempt may fail due to security considerations or resource limitations.

The deletion of applet package and contained instances operation must be atomic. If a reset or power fail occurs during the deletion process, it must result in either an unsuccessful deletion of the applet package and contained instances or a successfully completed deletion of the applet package and contained instances before any applet is selected on the card.

Following an unsuccessful deletion of the applet package and contained instances, any object or package that depends on the package continues to function unaffected. In other words, the functionality of all applets on the card remains the same as prior to the unsuccessful attempt.

Following a successful deletion of the applet package and contained instances, it shall not be possible to install another package which depends on the deleted package. Additionally, it shall be possible to re-install the same package (with exactly the same package AID) or an upgraded version of the deleted package onto the card.

The resources used by the applet package may be recovered for re-use.

Following a successful deletion of the applet package and contained instances, it shall not be possible to select any of the deleted applets, and no object owned by the deleted applets can be accessed by any applet currently on the card or by a new applet created in the future.

The resources used by the applet instances may be recovered for re-use.

The AID for the deleted applet instances may be re-assigned to new applet instances.

11.3.5 Applet Deletion Manager Privileges

Although an Applet Deletion Manager may be implemented as an applet, an Applet Deletion Manager will typically require access to features that are not available to "other" applets. For example, depending on the Java Card RE implementer's implementation, the Applet Deletion Manager will need to:

- Read and write directly to memory, bypassing the object system and/or standard security.
- Access objects owned by other applets or by the Java Card RE.
- Invoke non-entry point methods of the Java Card RE.

Again, it is up to each Java Card RE implementer to determine the Applet Deletion Manager implementation and supply such features in their Java Card RE implementations as necessary to support their Applet Deletion Manager. Java Card RE implementers are also responsible for the security of such features, so that they are not available to normal applets.

CHAPTER 12

API Constants

Some of the API classes don't have values specified for their constants in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1*. If constant values are not specified consistently by implementers of this *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1*, industry-wide interoperability is impossible. This chapter provides the required values for constants that are not specified in the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1*.

Class `javacard.framework.APDU`

```
public static final byte PROTOCOL_TYPE_MASK = (byte)0x0F;
public static final byte PROTOCOL_T0 = 0;
public static final byte PROTOCOL_T1 = 1;
public static final byte PROTOCOL_MEDIA_MASK = (byte)0xF0;
public static final byte PROTOCOL_MEDIA_DEFAULT = (byte)0x00;
public static final byte PROTOCOL_MEDIA_CONTACTLESS_TYPE_A =
    (byte)0x80;
public static final byte PROTOCOL_MEDIA_CONTACTLESS_TYPE_B =
    (byte)0x90;
public static final byte PROTOCOL_MEDIA_USB = (byte)0xA0;
public static final byte STATE_INITIAL = 0;
public static final byte STATE_PARTIAL_INCOMING = 1;
public static final byte STATE_FULL_INCOMING = 2;
public static final byte STATE_OUTGOING = 3;
public static final byte STATE_OUTGOING_LENGTH_KNOWN = 4;
public static final byte STATE_PARTIAL_OUTGOING = 5;
public static final byte STATE_FULL_OUTGOING = 6;
public static final byte STATE_ERROR_NO_T0_GETRESPONSE = (byte)-1;
public static final byte STATE_ERROR_T1_IFD_ABORT = (byte)-2;
public static final byte STATE_ERROR_IO = (byte) -3;
public static final byte STATE_ERROR_NO_T0_REISSUE = (byte) -4;
```

Class `javacard.framework.APDUException`

```
public static final short ILLEGAL_USE = 1;
public static final short BUFFER_BOUNDS = 2;
```

```
public static final short BAD_LENGTH = 3;
public static final short IO_ERROR = 4;
public static final short NO_T0_GETRESPONSE = 0xAA;
public static final short T1_IFD_ABORT = 0xAB;
public static final short NO_T0_REISSUE = 0xAC;
```

Interface javacard.framework.ISO7816

```
public final static short SW_NO_ERROR = (short)0x9000;
public final static short SW_BYTES_REMAINING_00 = 0x6100;
public final static short SW_WARNING_STATE_UNCHANGED = 0x6200;
public final static short SW_WRONG_LENGTH = 0x6700;
public final static short SW_LOGICAL_CHANNEL_NOT_SUPPORTED = 0x6881;
public final static short SW_SECURE_MESSAGING_NOT_SUPPORTED =
    0x6882;
public final static short SW_SECURITY_STATUS_NOT_SATISFIED = 0x6982;
public final static short SW_FILE_INVALID = 0x6983;
public final static short SW_DATA_INVALID = 0x6984;
public final static short SW_CONDITIONS_NOT_SATISFIED = 0x6985;
public final static short SW_COMMAND_NOT_ALLOWED = 0x6986;
public final static short SW_APPLET_SELECT_FAILED = 0x6999;
public final static short SW_WRONG_DATA = 0x6A80;
public final static short SW_FUNC_NOT_SUPPORTED = 0x6A81;
public final static short SW_FILE_NOT_FOUND = 0x6A82;
public final static short SW_RECORD_NOT_FOUND = 0x6A83;
public final static short SW_INCORRECT_P1P2 = 0x6A86;
public final static short SW_WRONG_P1P2  = 0x6B00;
public final static short SW_CORRECT_LENGTH_00 = 0x6C00;
public final static short SW_INS_NOT_SUPPORTED = 0x6D00;
public final static short SW_CLA_NOT_SUPPORTED = 0x6E00;
public final static short SW_UNKNOWN = 0x6F00;
public static final short SW_FILE_FULL = 0x6A84;
public final static byte OFFSET_CLA = 0;
public final static byte OFFSET_INS = 1;
public final static byte OFFSET_P1 = 2;
public final static byte OFFSET_P2 = 3;
public final static byte OFFSET_LC = 4;
public final static byte OFFSET_CDATA= 5;
public final static byte CLA_ISO7816 = 0x00;
public final static byte INS_SELECT = (byte) 0xA4;
public final static byte INS_EXTERNAL_AUTHENTICATE = (byte) 0x82;
```

Class javacard.framework.JCSystem

```
public static final byte NOT_A_TRANSIENT_OBJECT = 0;
public static final byte CLEAR_ON_RESET = 1;
public static final byte CLEAR_ON_DESELECT = 2;
public static final byte MEMORY_TYPE_PERSISTENT = 0;
public static final byte MEMORY_TYPE_TRANSIENT_RESET = 1;
public static final byte MEMORY_TYPE_TRANSIENT_DESELECT = 2;
```

Class javacard.framework.PINException

```
public static final short ILLEGAL_VALUE = 1;
```

Class javacard.framework.SystemException

```
public static final short ILLEGAL_VALUE = 1;
public static final short NO_TRANSIENT_SPACE = 2;
public static final short ILLEGAL_TRANSIENT = 3;
public static final short ILLEGAL_AID = 4;
public static final short NO_RESOURCE = 5;
public static final short ILLEGAL_USE = 6;
```

Class javacard.framework.TransactionException

```
public static final short IN_PROGRESS = 1;
public static final short NOT_IN_PROGRESS = 2;
public static final short BUFFER_FULL = 3;
public static final short INTERNAL_FAILURE = 4;
```

Class javacard.framework.service.Dispatcher

```
public static final byte PROCESS_NONE = (byte)0;
public static final byte PROCESS_INPUT_DATA = (byte)1;
public static final byte PROCESS_COMMAND= (byte)2;
public static final byte PROCESS_OUTPUT_DATA = (byte)3;
```

Class javacard.framework.service.RMIService

```
public static final byte DEFAULT_RMI_INVOKE_INSTRUCTION = 0x38;
```

Class javacard.framework.service.ServiceException

```
public static final short ILLEGAL_PARAM = 1;
public static final short DISPATCH_TABLE_FULL = 2;
public static final short COMMAND_DATA_TOO_LONG = 3;
public static final short CANNOT_ACCESS_IN_COMMAND = 4 ;
public static final short CANNOT_ACCESS_OUT_COMMAND =  5;
public static final short COMMAND_IS_FINISHED = 6;
public static final short REMOTE_OBJECT_NOT_EXPORTED = 7;
```

Class javacard.security.Checksum

```
public static final byte ALG_ISO3309_CRC16 = 1;
public static final byte ALG_ISO3309_CRC32 = 2;
```

Class javacard.security.CryptoException

```
public static final short ILLEGAL_VALUE = 1;
public static final short UNINITIALIZED_KEY = 2;
public static final short NO_SUCH_ALGORITHM = 3;
public static final short INVALID_INIT = 4;
public static final short ILLEGAL_USE = 5;
```

Class javacard.security.KeyAgreement

```
public static final byte ALG_EC_SVDP_DH = 1;
public static final byte ALG_EC_SVDP_DHC = 2;
```

Class javacard.security.KeyBuilder

```
public static final byte TYPE_DES_TRANSIENT_RESET = 1;
public static final byte TYPE_DES_TRANSIENT_DESELECT = 2;
public static final byte TYPE_DES = 3;
public static final byte TYPE_RSA_PUBLIC = 4;
public static final byte TYPE_RSA_PRIVATE = 5;
public static final byte TYPE_RSA_CRT_PRIVATE = 6;
public static final byte TYPE_DSA_PUBLIC = 7;
public static final byte TYPE_DSA_PRIVATE = 8;
public static final byte TYPE_EC_F2M_PUBLIC = 9;
public static final byte TYPE_EC_F2M_PRIVATE = 10;
public static final byte TYPE_EC_FP_PUBLIC = 11;
public static final byte TYPE_EC_FP_PRIVATE = 12;
public static final byte TYPE_AES_TRANSIENT_RESET = 13;
public static final byte TYPE_AES_TRANSIENT_DESELECT = 14;
public static final byte TYPE_AES = 15;
public static final short LENGTH_DES = 64;
public static final short LENGTH_DES3_2KEY = 128;
public static final short LENGTH_DES3_3KEY = 192;
public static final short LENGTH_RSA_512 = 512;
public static final short LENGTH_RSA_736 = 736;
public static final short LENGTH_RSA_768 = 768;
public static final short LENGTH_RSA_896 = 896;
public static final short LENGTH_RSA_1024 = 1024;
public static final short LENGTH_RSA_1280 = 1280;
public static final short LENGTH_RSA_1536 = 1536;
public static final short LENGTH_RSA_1984 = 1984;
public static final short LENGTH_RSA_2048 = 2048;
public static final short LENGTH_DSA_512 = 512;
public static final short LENGTH_DSA_768 = 768;
public static final short LENGTH_DSA_1024 = 1024;
public static final short LENGTH_EC_FP_112 = 112;
public static final short LENGTH_EC_F2M_113 = 113;
public static final short LENGTH_EC_FP_128 = 128;
public static final short LENGTH_EC_F2M_131 = 131;
public static final short LENGTH_EC_FP_160 = 160;
public static final short LENGTH_EC_F2M_163 = 163;
```

```
public static final short LENGTH_EC_FP_192 = 192;
public static final short LENGTH_EC_F2M_193 = 193;
public static final short LENGTH_AES_128= 128;
public static final short LENGTH_AES_192= 192;
public static final short LENGTH_AES_256= 256;
```

Class javacard.security.KeyPair

```
public static final byte ALG_RSA = 1;
public static final byte ALG_RSA_CRT = 2;
public static final byte ALG_DSA = 3;
public static final byte ALG_EC_F2M = 4;
public static final byte ALG_EC_FP = 5;
```

Class javacard.security.MessageDigest

```
public static final byte ALG_SHA = 1;
public static final byte ALG_MD5 = 2;
public static final byte ALG_RIPEMD160 = 3;
```

Class javacard.security.RandomData

```
public static final byte ALG_PSEUDO_RANDOM = 1;
public static final byte ALG_SECURE_RANDOM = 2;
```

Class javacard.security.Signature

```
public static final byte ALG_DES_MAC4_NOPAD = 1;
public static final byte ALG_DES_MAC8_NOPAD = 2;
public static final byte ALG_DES_MAC4_ISO9797_M1 = 3;
public static final byte ALG_DES_MAC8_ISO9797_M1 = 4;
public static final byte ALG_DES_MAC4_ISO9797_M2 = 5;
public static final byte ALG_DES_MAC8_ISO9797_M2 = 6;
public static final byte ALG_DES_MAC4_PKCS5 = 7;
public static final byte ALG_DES_MAC8_PKCS5 = 8;
public static final byte ALG_RSA_SHA_ISO9796 = 9;
public static final byte ALG_RSA_SHA_PKCS1 = 10;
public static final byte ALG_RSA_MD5_PKCS1 = 11;
public static final byte ALG_RSA_RIPEMD160_ISO9796 = 12;
public static final byte ALG_RSA_RIPEMD160_PKCS1 = 13;
public static final byte ALG_DSA_SHA = 14;
public static final byte ALG_RSA_SHA_RFC2409 = 15;
public static final byte ALG_RSA_MD5_RFC2409 = 16;
public static final byte ALG_ECDSA_SHA = 17;
public static final byte ALG_AES_MAC_128_NOPAD = 18;
public static final byte ALG_DES_MAC4_ISO9797_1_M2_ALG3 = 19;
public static final byte ALG_DES_MAC8_ISO9797_1_M2_ALG3 = 20;
public static final byte ALG_RSA_SHA_PKCS1_PSS = 21;
```

```
public static final byte ALG_RSA_MD5_PKCS1_PSS = 22;
public static final byte ALG_RSA_RIPEMD160_PKCS1_PSS = 23;
public static final byte MODE_SIGN = 1;
public static final byte MODE_VERIFY = 2;
```

Class javacardx.crypto.Cipher

```
public static final byte ALG_DES_CBC_NOPAD = 1;
public static final byte ALG_DES_CBC_ISO9797_M1 = 2;
public static final byte ALG_DES_CBC_ISO9797_M2 = 3;
public static final byte ALG_DES_CBC_PKCS5 = 4;
public static final byte ALG_DES_ECB_NOPAD = 5;
public static final byte ALG_DES_ECB_ISO9797_M1 = 6;
public static final byte ALG_DES_ECB_ISO9797_M2 = 7;
public static final byte ALG_DES_ECB_PKCS5 = 8;
public static final byte ALG_RSA_ISO14888 = 9;
public static final byte ALG_RSA_PKCS1 = 10;
public static final byte ALG_RSA_ISO9796 = 11;
public static final byte ALG_RSA_NOPAD = 12;
public static final byte ALG_AES_BLOCK_128_CBC_NOPAD = 13;
public static final byte ALG_AES_BLOCK_128_ECB_NOPAD = 14;
public static final byte ALG_RSA_PKCS1_OAEP = 15;
public static final byte MODE_DECRYPT = 1;
public static final byte MODE_ENCRYPT = 2;
```

Glossary

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

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

There is a unique AID for each package and a unique AID for each applet in the package. The package AID and the default AID for each applet defined in the package are specified in the `CAP` file. They are supplied to the converter when the `CAP` file is generated.

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

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

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

applet developer — refers to a person creating an applet using Java Card technology.

applet execution context — context of a package that contains currently active applet.

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

applet package — see *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 refers to whether a particular operation is atomic. Atomicity of data update 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.

basic logical channel logical channel 0, the only channel that is active at card reset. 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 to a type in a Java API package 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 file of the preexisting `CAP` file can link with the new `CAP` file without errors.

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

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

CAP file the `CAP` file is produced by the Converter and is the standard file format for the binary compatibility of the Java Card platform. A `CAP` file contains an executable binary representation of the classes of a Java API package. The `CAP` file also contains the `CAP` file components (see also *CAP file component*). The `CAP` files produced by the converter are contained in JAR files.

CAP file component a Java Card platform `CAP` file consists of a set of components which represent a Java API package. Each component describes a set of elements in the Java API package, 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

These components are optional: the Applet, Export, and Debug. The Applet component is included only if one or more Applets are defined in the package. The Export component is included only if classes in other packages may import elements in the package defined. The Debug component is optional. It contains all of the data necessary for debugging a package.

card session a card session begins with the insertion of the card into the CAD. The card is powered up, and exchanges streams of APDUs with the CAD. The card session ends when the card is removed from the CAD.

cast the explicit conversion from one data type to another.

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 ExportFile 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 entries. 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 package and Java Card RE. All objects owned by an applet belong to context of the applet's package.

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 applet instance that resides in a different package. The result of a context switch is a new currently active context.

Converter a piece of software that preprocesses all of the Java API class files that make up a package, and converts the package to a `CAP` file. The Converter also produces an export file.

currently active context when an object instance method is invoked, an owning context of this object becomes 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. Otherwise, it will be silently ignored by the Java Card virtual machine. The identifiers associated with the new component are recorded in the `custom_component` item of the `CAP` file's Directory component.

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

EEPROM an acronym for Electrically Erasable, Programmable Read Only Memory.

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

externally visible in the Java Card platform, "externally visible from a package" refers to any classes, interfaces, their constructors, methods and fields that can be accessed from another package according to the Java Language semantics, as defined by the *Java™ Language Specification*, and Java Card API package access control restrictions (see the *Java™ Language Specification*, section 2.2.1.1).

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

finalization the process by which a Java VM allows an unreferenced object instance to release non-memory resources (e.g. close/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 (i.e. there are no references to the object).

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

firewall see *applet firewall*.

flash memory a type of persistent mutable memory. It is more efficient in space and power that 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.

heap a common pool of free memory usable by a program. A part of the computer's memory used for dynamic memory allocation, in which blocks of memory are used in an arbitrary order. The Java Card virtual machine's heap is not required to be garbage collected. Objects allocated from the heap will not necessarily be 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 are also known as non-static fields.

instantiation in object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class's constructor function.

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

internally visible items which are not externally visible. These items are not described in a package's export file, but some such items use private tokens to represent internal references. See also *externally visible*.

JAR file a file format used for aggregating many files into one.

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

Java Card Runtime Environment (Java Card RE) consists of the Java Card virtual machine, the 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 entry point objects objects owned by the Java Card RE context that contain entry point methods. These methods can be invoked from any context and allow non-privileged users (applets) to request privileged Java Card RE system services. Java Card RE entry point objects 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 re-use. Examples of these objects are APDU objects and all Java Card RE-owned exception objects.

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

JDK is an acronym for Java Development Kit. The JDK is a Sun Microsystems, Inc. product that provides the environment required for software development in the Java programming language. The JDK is available for a variety of platforms, for example Sun Solaris and Microsoft Windows.

library package — a library package is a Java API package that does not contain any non-abstract classes which extend the class `javacard.framework.Applet`. An applet package contains one or more non-abstract classes which extend the `javacard.framework.Applet` class.

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 can't be used outside the method.

logical channel — a logical channel, as seen at the card edge, works as a logical link to an 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 7816 Specification, Part 4*. (`http://www.iso.org`).

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

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

method — the name given to 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 at the same time. They can also accept other applets belonging to the same package being selected simultaneously.

multiselected applet — an applet instance that is selected (that is, the active applet instance) on more than one logical channel simultaneously.

namespace — a set of names in which all names are unique.

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

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 owner — the applet instance within the currently active context when the object is instantiated. An object can be owned by an applet instance, or by the Java Card RE.

objects — in object-oriented programming, are unique instances 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 on which an APDU command is issued.

owning context — the context in which an object is instantiated or created.

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

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

PIX — see **AID**.

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

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

remote interface — an interface 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. These constraints are as a result of the Java Card platform language subset and other feature limitations.

remote methods — the methods of a remote interface.

remote object — an object whose remote methods can be invoked remotely from the CAD client. A remote object is described by one or more remote interfaces.

RFU — acronym for “Reserved for Future Use”.

RID — see **AID**.

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

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

runtime environment see *Java Card Runtime Environment (Java Card RE)*.

shareable interface defines a set of shared interface methods. These interface methods can be invoked from an applet 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 which 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.

terminal a Card Acceptance Device that is typically a computer in its own right and can integrate a card reader as one of its components. In addition to being a smart card reader, a terminal can process data exchanged between itself and the smart card.

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, the thread is suspended or destroyed.

The Java Card virtual machine can support only a single thread of execution. Java Card 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 CAD 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.

verification a process performed on a `CAP` file which ensures that the binary representation of the package is structurally correct.

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

Index

D

E

F

G

I

J

L

M

O

P

R

S

T

V