Java Card ™ 2.1 Runtime Environment (JCRE) Specification

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300

Final Revision 1.0, February 24, 1999

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 software programming to the resource-constrained world of smart cards.

This document is a specification of the Java Card 2.1 Runtime Environment (JCRE). A vendor of a Java Card-enabled device provides an implementation of the JCRE. A JCRE implementation within the context of this specification refers to a vendor's implementation of the Java Card Virtual Machine (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. Applets written for the Java Card platform are referred to as Java Card applets.

Who Should Use This Specification?

This specification is intended to assist JCRE implementers in creating an implementation, developing a specification to extend the Java Card technology specifications, or in creating an extension to the Java Card Runtime Environment (JCRE). 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, "The Scope and Responsibilities of the JCRE," gives an overview of the services required of a JCRE implementation.

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, "Selection," describes how the JCRE 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, "API Topics," describes API functionality required of a JCRE but not completely specified in the *Java Card 2.1 API Specification*.

Chapter 9, "Virtual Machine Topics," describes virtual machine specifics.

Chapter 10, "Applet Installer," provides an overview of the Applet Installer and JCRE required behavior.

Chapter 11, "API Constants," provides the numeric value of constants that are not specified in the *Java Card 2.1 API Specification*.

Glossary is a list of words and their definitions to assist you in using this book.

Related Documents and Publications

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

- *Java Card 2.1 API Specification*, Sun Microsystems, Inc.
- *Java Card 2.1 Virtual Machine Specification,* Sun Microsystems, Inc.
- *Java Card Applet Developer's Guide*, Sun Microsystems, Inc.
- *The Java Language Specification* by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1.
- *The Java Virtual Machine Specification* (Java Series) by Tim Lindholm and Frank Yellin. Addison-Wesley, 1996, ISBN 0-201-63452-X.
- *The Java Class Libraries: An Annotated Reference (Java Series)* by Patrick Chan and Rosanna Lee. Addison-Wesley, two volumes, ISBN: 0201310023 and 0201310031.
- ISO 7816 Specification Parts 1-6.
- EMV '96 Integrated Circuit Card Specification for Payment Systems.

1. Introduction

The Java Card 2.1 Runtime Environment (JCRE) 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 Java Card 2.1 Environment (JCRE) Specification, specifies the JCRE functionality required by the Java Card technology. Any implementation of Java Card technology shall provide this necessary behavior and environment.

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 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 JCRE is initialized. The framework objects created by the JCRE exist for the lifetime of the Virtual Machine. Because the execution lifetime of the Virtual Machine and the JCRE 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 JCRE implementer shall make an object persistent when:

- The `Applet.register` method is called. The JCRE stores a reference to the instance of the applet object. The JCRE 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 JCRE's internal data structures.

3. Java Card Applet Lifetime

For the purposes of this specification, a Java Card applet's lifetime begins at the point that it has been correctly loaded into card memory, linked, and otherwise prepared for execution. (For the remainder of this specification, *applet* refers to an applet written for the Java Card platform.) Applets registered with the `Applet.register` method exist for the lifetime of the card. The JCRE 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 JCRE 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 JCRE for each applet instance created. The JCRE shall not call the applet's constructor directly.

3.1 The Method `install`

When the `install(byte[], short, byte)` method is called, no objects of the applet exist. 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 JCRE shall perform all cleanup when it regains control. That is, all persistent objects shall be returned to the state they had prior to calling the `install` method. If the installation is successful, the JCRE can mark the applet as available for selection.

3.2 The Method `select`

Applets remain in a suspended state until they are explicitly selected. Selection occurs when the JCRE receives a SELECT APDU in which the name data matches the AID of the applet. Selection causes an applet to become the currently selected applet.

Prior to calling SELECT, the JCRE shall deselect the previously selected applet. The JCRE indicates this to the applet by invoking the applet's `deselect` method.

The JCRE informs the applet of selection by invoking its `select()` method.

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 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 APDU command exactly like it processes any other APDU command. It can respond to the SELECT APDU with data (see the `process` method for details), or it can flag errors by throwing an `ISOException` with the appropriate SW (returned status word). The SW 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 APDU command.

If the applet declines to be selected, the JCRE will return an APDU response status word of `ISO7816.SW_APPLET_SELECT_FAILED` to the CAD. Upon selection failure, the JCRE state is set to indicate that no applet is selected. (See section 4.2 for more details).

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

3.3 The Method `process`

All APDUs are received by the JCRE, which passes an instance of the APDU class to the `process(APDU)` method of the currently selected applet.

Note – A SELECT 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 JCRE automatically appends 0x9000 as the completion response SW to any data already sent by the applet.

At any time during `process`, the applet may throw an `ISOException` with an appropriate SW, in which case the JCRE catches the exception and returns the SW to the CAD.

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

3.4 The Method `deselect`

When the JCRE receives a SELECT APDU command in which the name matches the AID of an applet, the JCRE calls the `deselect()` method of the currently selected applet. This 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 JCRE, but the applet is deselected.

3.5 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 JCRE 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 that was selected when power was lost (or reset occurred) becomes implicitly deselected. (In this case the `deselect` method is not called.)
- If the JCRE implements default applet selection (see section 4.1), the default applet is selected as the currently selected applet, and the default applet's `select` method is called. Otherwise, the JCRE sets its state to indicate that no applet is selected.

4. Selection

Cards receive requests for service from the CAD in the form of APDUs. The SELECT APDU is used by the JCRE to designate a *currently selected applet.* Once selected, an applet receives all subsequent APDUs until the applet becomes deselected.

There is no currently selected applet when either of the following occurs:

- The card is reset and no applet has been pre-designated as the *default applet*.
- A SELECT command fails when attempting to select an applet via its `select` method .

4.1 The Default Applet

Normally, applets become selected only via a successful SELECT command. However, some smart card CAD applications require that there be a default applet that is implicitly selected after every card reset. The behavior is:

1. After card reset (or power on, which is a form of reset) the JCRE performs its initializations and checks to see if its internal state indicates that a particular applet is the default applet. If so, the JCRE makes this applet the currently selected applet, and the applet's `select` method is called. If the applet's `select` method throws an exception or returns `false`, then the JCRE sets its state to indicate that no applet is selected. (The applet's `process` method is not called during default applet selection because there is no SELECT APDU.) When a default applet is selected at card reset, it shall not require its `process` method to be called.

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

If a default applet was successfully selected, then APDU commands can be sent directly to this applet. If a default applet was not selected, then only SELECT commands for applet selection can be processed.

The mechanism for specifying a default applet is not defined in the Java Card 2.1 API. It is a JCRE implementation detail and is left to the individual JCRE implementers.

4.2 SELECT Command Processing

The SELECT APDU command is used to select an applet. Its behavior is:

1. The SELECT APDU is always processed by the JCRE regardless of which, if any, applet is active.

2. The JCRE searches the internal applet table which lists all successfully installed applets on the card for a matching AID. The JCRE shall support selecting an applet where the full AID is present in the SELECT command.

   JCRE implementers are free to enhance their JCRE to support other selection criterion. An example of this is selection via partial AID match as specified in ISO 7816-4. The specific requirements are as follows:

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

---

   a) Applet SELECT command uses CLA=0x00, INS=0xA4.

   b) Applet SELECT command uses "Selection by DF name". Therefore, P1=0x04.

   c) Any other value of P1 implies that is not an applet select. The APDU is processed by the currently selected applet.

   d) JCRE shall support exact DF name (AID) selection (i.e. P2=%b0000xx00). (b4,b3* are don't care).

   e) All other partial DF name SELECT options (b2,b1*) are JCRE implementation dependent.

   f) All file control information option codes (b4,b3*) shall be supported by the JCRE and interpreted and processed by the applet.

3. If no AID match is found:

   a. If there is no currently selected applet, the JCRE responds to the SELECT command with status code 0x6999 (SW_APPLET_SELECT_FAILED).

   b. Otherwise, the SELECT command is forwarded to the currently selected applet's `process` method. A context switch into the applet's context occurs at this point. (Context of an applet is defined in section 6.1.1.) Applets may use the SELECT APDU command for their own internal SELECT processing.

4. If a matching AID is found, the JCRE prepares to select the new applet. If there is an currently selected applet, it is deselected via a call to its `deselect` method. A context switch into the deselected applet's context occurs at this point. The JCRE context is restored upon exit from `deselect`.

5. The JCRE now clears the fields of all CLEAR_ON_DESELECT transient objects (see section 5.1) owned by the applet being deselected.

6. The JCRE sets the new currently selected applet. The new applet is selected via a call to its select method, and a context switch into the new applet's context occurs

   a. If the applet's `select` method throws an exception or returns `false`, then the JCRE state is set so that no applet is selected. The JCRE responds to the SELECT command with status code 0x6999 (SW_APPLET_SELECT_FAILED).

b. The new currently selected applet's `process` method is then called with the SELECT APDU as an input parameter. A context switch into the applet's context occurs.

Notes –

If there is no matching AID, the SELECT command is forwarded to the currently selected applet (if any) for processing as a normal applet APDU command.

If there is a matching AID and the SELECT command fails, the JCRE always enters the state where no applet is selected.

If the matching AID is the same as the currently selected applet, the JCRE still goes through the process of deselecting the applet and then selecting it. Reselection could fail, leaving the card in a state where no applet is selected.

4.3 Non-SELECT Command Processing

When a non-SELECT APDU is received and there is no currently selected applet, the JCRE shall respond to the APDU with status code 0x6999 (SW_APPLET_SELECT_FAILED).

When a non-SELECT APDU is received and there is a currently selected applet, the JCRE invokes the `process` method of the currently selected applet passing the APDU as a parameter. This causes a context switch from the JCRE context into the currently selected applet's context. When the `process` method exits, the VM switches back to the JCRE context. The JCRE sends a response APDU and waits for the next command APDU.

5. Transient Objects

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

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).
- 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. 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 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 regardless of whether the SELECT command:

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

6. Applet Isolation and Object Sharing

Any implementation of the JCRE 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 JCRE 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 Contexts and Context Switching

Firewalls essentially partition the Java Card platform's object system into separate protected object spaces called *contexts*. The firewall is the boundary between one context and another. The JCRE shall allocate and manage a *context* for each applet that is installed on the card. (But see section 6.1.1.2 below for a discussion of group contexts.)

In addition, the JCRE maintains its own *JCRE 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.

At any point in time, there is only one *active context* within the VM. (This is called the *currently active 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.

When certain well-defined conditions are met during the execution of invoke-type bytecodes as described in section 6.2.8, the VM performs a *context switch*. The previous context 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. Context switches only occur during invocation of and return from certain methods, as well as during exception exits from those methods (see 6.2.8).

During a context-switching method invocation, an additional piece of data, indicating the currently active context, is pushed onto the return stack. This context is restored when the method is exited.

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

6.1.1.1 Group Contexts

Usually, each instance of a Java Card applet defines a separate context. But with Java Card 2.1 technology, the concept of *group context* is introduced. If more than one applet is contained in a single Java package, they share the same context. Additionally, all instances of the same applet class share the same context. In other words, there is no firewall between two applet instances in a group context.

The discussion of contexts and context switching above in section 6.1.1 assumes that each applet instance is associated with a separate context. In Java Card 2.1 technology, contexts are compared to enforce the firewall, and the instance AID is pushed onto the stack. Additionally, this happens not only when the context switches, but also when control switches from an object owned by one applet instance to an object owned by another instance within the same package.

6.1.2 Object Ownership

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 is owned by an applet instance, or by the JCRE.

6.1.3 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`, `sgetfield_s_this`, etc.

6.1.4 Firewall Protection

The 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, but if the object is owned by an applet in another context, the firewall ensures security.

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 *Java Card 2.1 JCRE Specification* 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.

JCRE 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.

6.1.5 Static Fields and Methods

It should also be noted that classes are not owned by contexts. 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.

6.1.5.1 Optional static access checks

The JCRE 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

To enable applets to interact with each other and with the JCRE, 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 2.1 API and are discussed in the following sections:

- JCRE Entry Point Objects
- Global Arrays
- JCRE Privileges
- Shareable Interfaces

6.2.1 JCRE 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 2.1 API, this is accomplished using *JCRE Entry Point Objects*. These are objects owned by the JCRE 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 JCRE context is performed. These methods are the gateways through which applets request privileged JCRE system services.

There are two categories of JCRE Entry Point Objects :

- Temporary JCRE Entry Point Objects

  Like all JCRE Entry Point Objects, methods of temporary JCRE 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 JCRE 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 JCRE owned exception objects are examples of temporary JCRE Entry Point Objects.

- Permanent JCRE Entry Point Objects

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

  JCRE owned AID instances are examples of permanent JCRE Entry Point Objects.

The JCRE 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 JCRE Entry Point Objects.
- Designating JCRE 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 JCRE context.

Only the JCRE itself can designate Entry Point Objects and whether they are temporary or permanent. JCRE implementers are responsible for implementing the mechanism by which JCRE 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 JCRE 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 JCRE 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 JCRE itself can designate global arrays. Because applets cannot create them, no API methods are defined. JCRE 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 2.1 API are the APDU buffer and the byte array input parameter (`bArray`) to the applet's `install` method.

Note – Because of its global status, the *Java Card 2.1 API Specification* specifies that the APDU buffer is cleared to zeroes whenever an applet is selected, before the JCRE 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 JCRE Privileges

Because it is the "system" context, the JCRE 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 JCRE context is allowed to invoke any of the methods of X. During such an invocation, a context switch occurs from the JCRE context to the context of the applet that owns X.

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

The JCRE context is the currently active context when the VM begins running after a card reset. The JCRE 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 new feature in the Java Card 2.1 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 or not 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 current 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 JCRE shall return the instance AID of the applet instance active at the time of the last context switch.

6.2.5.1 The JCRE Context

The JCRE context does not have an AID. If an applet calls the `getPreviousContextAID` method when the context of the applet was entered directly from the JCRE 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 Shareable Interface

Interfaces extending the `Shareable` tagging interface have this special property: calls to the interface methods take place across Java Card'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. In order 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 JCRE 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 JCRE 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 JCRE shall implement this method to behave as follows:

1. The JCRE 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 JCRE 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 bytecodes is executed:

```
getstatic, putstatic
```

An object is *accessed* when one of the following Java 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 may be implemented in the Java Card VM, such as `getfield_b`, `sgetfield_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 JCRE 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 JCRE 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 JCRE 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 JCRE 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 JCRE 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 JCRE 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. Context is switched to the object owner's context.
- Otherwise, if the object is designated a JCRE Entry Point Object, then access is allowed. Context is switched to the object owner's context (shall be JCRE).
- Otherwise, if JCRE 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 JCRE 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'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 JCRE 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 JCRE Entry Point Object, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed.
- Otherwise, access is denied.

6.2.8.8 Accessing Class Instance Objects

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 JCRE Entry Point Object, then access is allowed.
- Otherwise, if the JCRE 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 JCRE 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 an instance of (instanceof) an interface that extends the `Shareable` interface, then access is allowed.
- Otherwise, if the JCRE is the currently active context, then access is allowed.

- Otherwise, access is denied.

6.3 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 same context as the owner of the object (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, 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, the JCRE shall throw a `java.lang.SecurityException`.

Applets that are part of the same package share the same group 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 within the same package shall be accessible when any of the applet instances in this package is the currently selected applet.

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 JCRE 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 JCRE regains programmatic control upon return from the applet's `select`, `deselect`, `process` 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 JCRE). 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 JCRE 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 JCRE when it reinitializes the card after recovering from the power loss, reset, or failure. The JCRE determines which of those objects (if any) were conditionally updated, and restores them.

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

7.6 Aborting a Transaction

Transactions can be aborted either by an applet or by the JCRE.

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 JCRE

If an applet returns from the `select`, `deselect`, `process`, or `install` methods with a transaction in progress, the JCRE automatically aborts the transaction. If a return from any of `select`, `deselect`, `process` or `install` methods occurs with a transaction in progress, the JCRE acts as if an exception was thrown.

7.6.3 Cleanup Responsibilities of the JCRE

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 JCRE shall ensure that a reference to an object created during the aborted transaction is equivalent to a `null` reference.

7.7 Transient Objects

Only updates to persistent objects participate in the transaction. Updates to transient objects 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 6.1.1), updates to persistent data continue to be conditional in the new context until the transaction is committed or aborted.

8. API Topics

The topics in this chapter complement the requirements specified in the *Java Card 2.1 API Specification*.

8.1 Resource Use within the API

Unless specified in the *Java Card 2.1 API Specification*, 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.

8.2 Exceptions thrown by API classes

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

8.3 Transactions within the API

Unless explicitly called out in the API descriptions, implementation of the Java Card 2.1 API methods shall not initiate or otherwise alter the state of a transaction in progress. Even if a transaction is in progress, updates to implementation persistent state within the API need not be conditional unless specifically called out by the API method.

8.4 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 JCRE may support T=0 or T=1 transport protocols or both.

8.4.1 T=0 specifics for outgoing data transfers

For compatibility with legacy CAD/terminals that do not support block chained mechanisms, the APDU Class allows mode selection via the `setOutgoingNoChaining` method.

8.4.1.1 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:

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`.

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.

8.4.1.2 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.

8.4.1.3 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, the `sendBytes` or the `sendBytesLong` methods shall throw an `APDUException` with reason code `NO_T0_GETRESPONSE`.

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 `NO_T0_GETRESPONSE` exception has been thrown, the JCRE shall discard the response status in its reason code. The JCRE shall restart APDU processing with the newly received command and resume APDU dispatching.

8.4.2 T=1 specifics for outgoing data transfers

8.4.2.1 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`.

8.4.2.2 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 anytime. 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).

8.4.2.2.1 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 JCRE shall discard the response status in its reason code. The JCRE shall restart APDU processing with the newly received command, and resume APDU dispatching.

8.4.3 T=1 specifics for incoming data transfers

8.4.3.1 Incoming transfers using chaining

8.4.3.1.1 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 JCRE shall discard the response status in its reason code. The JCRE shall restart APDU processing with the newly received command, and resume APDU dispatching.

8.5 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`

`javacardx.crypto.Cipher`

An implementation of the JCRE may implement 0 or more of the algorithms listed in the *Java Card 2.1 API Specification*. 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 JCRE 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.

8.6 JCSystem Class

In Java Card 2.1, the `getVersion` method shall return (short) 0x0201.

9. Virtual Machine Topics

The topics in this chapter detail virtual machine specifics.

9.1 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. Applet Installer

Applet installation on smart cards using Java Card technology is a complex topic. The design of the *Java Card 2.1 API Specification* is intended to give JCRE 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 without knowing the implementation details of a particular installer.

This specification defines the concept of an Installer and specifies minimal installation requirements in order to achieve interoperability across a wide range of possible Installer implementations.

The Applet Installer is an optional part of the Java Card 2.1 Environment (JCRE) Specification. That is, an implementation of the JCRE 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.

10.1 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 command. Once selected, the Installer behaves in much the same way as any other applet:

- It receives all APDUs just like any other selected 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 (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.

10.1.1 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 when a non-Installer applet is selected nor when no applet is selected.

Obviously, a JCRE 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.

But a JCRE 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 JCRE implementer has the freedom to provide some other mechanism by which APDUs are delivered to the Installer code module.

10.1.2 Installer AID

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

10.1.3 Installer APDUs

The Java Card 2.1 API does not specify any APDUs for the Installer. JCRE 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 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 *Java Card 2.1 API Specification* does not specify the details of the CAD installation program nor the APDUs passed between it and the Installer.

10.1.4 Installer Behavior

JCRE 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 JCRE shall guarantee that an applet will *not* be deemed successfully installed if:

- the applet package must link with 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 *Java Card 2.1 Virtual Machine Specification.*

- the applet's `install` method throws an exception before successful return from the `Applet.register` method (see section 3.1).

10.1.5 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 JCRE 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 JCRE.
- Invoke non-entry point methods of the JCRE.
- Be able to invoke the `install` method of a newly installed applet.

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

10.2 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 JCRE 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 during applet class initialization (<clinit>) methods are also owned by the 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 32 bytes. And for security reasons, the `bArray` parameter is zeroed after the return (just as the APDU buffer is zeroed on return from an applet's `process` method.)

10.2.1 Installation Parameters

Other than the maximum size of 32 bytes, the Java Card 2.1 API does not specify anything about the contents of the global byte array installation parameter. This is fully defined by the applet designer and can be in any format desired. In addition, these installation parameters are intended to be opaque to the Installer.

JCRE implementers should design their Installers so that it is possible for an installation program running in a CAD to specify an arbitrary byte array to be delivered to the Installer. The Installer simply forwards this byte array to the target applet's `install` method in the `bArray` parameter. A typical implementation might define a JCRE implementer-proprietary APDU command that has the semantics "call the applet's `install` method passing the contents of the accompanying byte array."

11. API Constants

Some of the API classes don't have values specified for their constants in the *Java Card 2.1 API Specification*. If constant values are not specified consistently by implementers of this Java Card 2.1 Environment (JCRE) Specification, industry-wide interoperability is impossible. This chapter provides the required values for constants that are not specified in the *Java Card 2.1 API Specification*.

Class javacard.framework.APDU

```
public static final byte PROTOCOL_T0 = 0;
public static final byte PROTOCOL_T1 = 1;
```

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;
```

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_WRONG_LENGTH = 0x6700;
public static final 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;
```

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;
```

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.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.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 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_768 = 768;
public static final short LENGTH_RSA_1024 = 1024;
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;
```

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 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 MODE_DECRYPT = 1;
public static final byte MODE_ENCRYPT = 2;
```

Glossary

AID is an acronym for Application IDentifier as defined in ISO 7816-5.

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

API is 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 a Java Card applet using the Java Card technology specifications.

Applet firewall is the mechanism in the Java Card technology by which the VM prevents an applet in one context from making unauthorized accesses to objects owned by an applet in another context or the JCRE context, and reports or otherwise addresses the violation.

Atomic operation is an operation that either completes in its entirety (if the operation succeeds) or no part of the operation completes at all (if the operation fails).

Atomicity refers to whether a particular operation is atomic or not and is necessary for proper data recovery in cases in which power is lost or the card is unexpectedly removed from the CAD.

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

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

Cast is the explicit conversion from one data type to another.

cJCK is the test suite to verify the compliance of the implementation of the Java Card Technology specifications. The cJCK uses the JavaTest tool to run the test suite.

Class is the prototype for an object in an object-oriented language. A class may also be considered a set of objects that share a common structure and behavior. The structure of a class is determined by the class variables that represent the state of an object of that class and the behavior is given by a set of methods associated with the class.

Classes are related in a class hierarchy. One class may be a specialization (a subclass) of another (its superclass), it may have reference to other classes, and it may use other classes in a client-server relationship.

Context (See Applet execution context.)

Currently active context. The JCRE keeps track of the currently active Java Card context. When a virtual method is invoked on an object, and a context switch is required and permitted, the currently active context is

changed to correspond to the context of the applet that owns the object. When that method returns, the previous context is restored. Invocations of static methods have no effect on the currently active context. The currently active context and sharing status of an object together determine if access to an object is permissible.

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

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

Firewall (see Applet Firewall).

Framework is the set of classes that implement the API. This includes core and extension packages. Responsibilities include dispatching of APDUs, applet selection, managing atomicity, and installing applets.

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

Instance variables, also known as fields, represent a portion of an object's internal state. Each object has its own set of instance variables. Objects of the same class will have the same instance variables, but each object can have different values.

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.

JAR is an acronym for Java Archive. JAR is a platform-independent file format that combines many files into one.

Java Card Runtime Environment (JCRE) consists of the Java Card Virtual Machine, the framework, and the associated native methods.

JC21RI is an acronym for the Java Card 2.1 Reference Implementation.

JCRE implementer refers to a person creating a vendor-specific implementation using the Java Card API.

JCVM is an acronym for the Java Card Virtual Machine. The JCVM is the foundation of the OP card architecture. The JCVM executes byte code and manages classes and objects. It enforces separation between applications (firewalls) and enables secure data sharing.

JDK is an acronym for Java Development Kit. The JDK is a Sun Microsystems, Inc. product that provides the environment required for programming in Java. The JDK is available for a variety of platforms, but most notably Sun Solaris and Microsoft Windows®.

Method is the name given to a procedure or routine, associated with one or more classes, in object-oriented languages.

Namespace is a set of names in which all names are unique.

Object-Oriented is 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.

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.

Package is a namespace within the Java programming language and can have classes and interfaces. A package is the smallest unit within the Java programming language.

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.

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.

Transaction is 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 values 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.