Application Programming Notes

Java Card™ Platform, Version 2.2.2

Sun Microsystems, Inc.
www.sun.com

3-11-06

Contents

Preface

This book contains tips and guidelines for applet developers and for developers of vendor-specific frameworks.

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

The Java Card API is compatible with international standards such as ISO7816 and industry-specific standards such as Europay, Master Card, Visa (EMV).

Who Should Use This Book

This book is for applet developers using the *Application Programming Interface for the Java Card Platform, Version 2.2.2* to implement applet management, multiselectable applets, logical channels, APDU I/O, Remote Method Invocation (RMI), and extended APDUs for the Java Card platform.

This book is also for developers who are considering creating a vendor-specific framework based on version 2.2.2 of the Java Card technology specifications.

Before You Read This Book

Before reading this guide, become familiar with the Java programming language, object-oriented design, the Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java and Java Card technology is the Sun Microsystems, Inc. web site, located at

`http://java.sun.com`

You must also be familiar with the development tools released with version 2.2.2 of the Java Card platform. For information on these tools, see the *Development Kit User's Guide, Java Card Platform, Version 2.2.2*.

How This Book Is Organized

Chapter 1 describes how to perform object deletion, applet deletion, and package deletion on the Java Card platform.

Chapter 2 describes how to create and use applets that can be selected for use on multiple channels on the Java Card platform.

Chapter 3 describes how to use APDU I/O to create and use applets.

Chapter 4 describes how to develop applications that use RMI on the Java Card platform.

Chapter 5 describes how to handle extended APDU functionality on the Java Card platform.

Related Books

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

- *Development Kit User's Guide for the Java Card Platform, Version 2.2.2.*
- *Application Programming Interface for the Java Card Platform, Version 2.2.2.*
- *Virtual Machine Specification for the Java Card Platform, Version 2.2.2.*

- *Runtime Environment Specification for the Java Card Platform, Version 2.2.2.*
- *Java Card Technology for Smart Cards* by Zhiqun Chen (Addison-Wesley, 2000).
- *Off-Card Verifier for the Java Card Platform, Version 2.2.1, White Paper* (Sun Microsystems, Inc., 2003) , Sun Microsystems, Inc.
- *The Java Programming Language (Java Series), Second Edition* by Ken Arnold and James Gosling (Addison-Wesley, 1998).
- *The Java Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999).
- *The Java Class Libraries: An Annotated Reference, Second Edition (Java Series)* by Patrick Chan, Rosanna Lee and Doug Kramer (Addison-Wesley, 1999).
- *ISO 7816 Specification* Parts 1-6.

Version 2.2.2 of the *Development Kit User's Guide* and the Java Card specifications are included in this development kit for the binary release. You can also download the identical specifications bundle separately from the Sun Microsystems' web site at

`http://java.sun.com/products/javacard`

Typographic Conventions

The following table lists the typographic conventions used in this book.

TABLE P-1 Typographic Convertions For This Book

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`
		1. **Run** `cref` **in a new window.**

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Accessing Sun Documentation Online

Access Java platform technical documentation on the web at the Java Developer Connection™ program web site at:

`http://java.sun.com/reference/`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. Email your comments to us at `docs@java.sun.com`.

CHAPTER 1

Using Object, Package and Applet Deletion

This chapter describes how to use the object deletion mechanism and the package and applet deletion features of the Java Card platform.

Object Deletion Mechanism

The object deletion mechanism on the Java Card platform reclaims memory that is being used by “unreachable” objects. For an object to be unreachable, neither a static nor an object field can point to an object. An applet object is reachable until successfully deleted.

The object deletion mechanism on the Java Card platform is not like garbage collection in standard Java technology applications due to space and time constraints. The amount of available RAM on the card is limited. In addition, because object deletion mechanism is applied to objects stored in persistent memory, it must be used sparingly. EEPROM writes are very time-consuming operations and only a limited number of writes can be performed on a card. Due to these limitations, the object deletion mechanism in Java Card technology is not automatic. It is performed only when an applet requests it. Use the object deletion mechanism sparingly and only when other Java Card technology-based facilities are cumbersome or inadequate.

The object deletion mechanism on the Java Card platform is not meant to change the programming style in which programs for the Java Card platform are written.

Requesting the Object Deletion Mechanism

Only the runtime environment for the Java Card platform (Java Card Runtime Environment or Java Card RE) can start the object deletion mechanism, although any applet on the card can request it. The applet requests the object deletion mechanism with a call to the `JCSystem.requestObjectDeletion()` method.

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

```
/**
* The following method updates the buffer size by removing
* the old buffer object from the memory by requesting
* object deletion and creates a new one with the
* required size.
*/
void updateBuffer(byte requiredSize){
     try{
         if(buffer != null && buffer.length == requiredSize){
             //we already have a buffer of required size
             return;
         }
         JCSystem.beginTransaction();
         byte[] oldBuffer = buffer;
         buffer = new byte[requiredSize];
         if (oldBuffer != null)
             JCSystem.requestObjectDeletion();
         JCSystem.commitTransaction();
     }catch(Exception e){
         JCSystem.abortTransaction();
     }
}
```

Object Deletion Mechanism Usage Guidelines

Do not confuse the object deletion mechanism on the Java Card platform with garbage collection in the standard Java programming language. The following guidelines describe the possible scenarios when the object deletion mechanism might or might not be used:

- When throwing exceptions, avoid creating new exception objects and relying on the object deletion mechanism to perform cleanup. Try to use existing exception objects.
- Similarly, try not to create objects in method or block scope. This is acceptable in standard Java technology applications, but is an incorrect use of the object deletion mechanism in Java Card technology-based applications.

- Use the object deletion mechanism when a large object, such as a certificate or key, must be replaced with a new one. In this case, instead of updating the old object in a transaction, create a new object and update its pointer within the transaction. Then, use the object deletion mechanism to remove the old object.
- Use the object deletion mechanism when object resizing is required, as shown in the example in "Requesting the Object Deletion Mechanism" on page 2.

Package and Applet Deletion

Version 2.2.2 of the Java Card platform provides the ability to delete package and applet instances from the card's memory. Requests for deletion are sent in the form of an APDU from the terminal to the smart card. Requests to delete an applet or package cannot be sent from an applet on the card.

In version 2.2.2 of the Java Card platform, the installer deletes packages and applets. Once the installer is selected, it can receive requests from the terminal to delete packages and applets. The following sections describe programming guidelines that will help your packages and applets to be more easily removed.

Developing Removable Packages

Package deletion refers to removing all of a package's code from the card's memory. To be eligible for deletion, nothing on the card can have dependencies on the package to be deleted, including the following:

- Packages that are dependent on the package to be deleted
- Applet instances or objects that either belong to the package, or that belong to a package that depends on the package to be deleted

Package deletion will not succeed if any of the following conditions exist:

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

To ensure that a package can be removed from the card easily, avoid writing and downloading other packages that might be dependent on the package. If there are other packages on the card that depend on this package, then you must remove all of the dependent packages before you can remove this package from the card memory.

Writing Removable Applets

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

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

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

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

Using the `AppletEvent.uninstall` Method

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

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

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

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

To ensure that the applets are deleted, implement the `uninstall` method defensively. Write your applet with these guidelines in mind:

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

The following example shows such an implementation:

```
public class TestApp1 extends Applet implements AppletEvent{

    // field set to true after uninstall
    private boolean disableApp = false;

    ...
    public void uninstall(){
        if (!disableApp){
            JCSystem.beginTransaction();  //to protect against tear
            disableApp = true;            //mark as uninstalled
            TestApp2SIO.removeDependency();
            JCSystem.commitTransaction();
        }
    }

    public boolean select(boolean appInstAlreadyActive) {
        // refuse selection if in uninstalled state
        if (disableApp) return false;
        return true;
    }
    ...

}
```

CHAPTER **2**

Working with Logical Channels

Version 2.2.2 of the Java Card platform has the ability to support up to twenty logical channels per active interface. This gives an ISO-7816-4:2005-compliant terminal the ability to open as many as twenty sessions into the smart card, one session per logical channel. Logical channels allow the concurrent execution of multiple applications on the card, allowing a terminal to handle different tasks at the same time.

Applets written for version 2.1 of the Java Card platform still work correctly, but they are unaware of logical channel support. In contrast, applets written for version 2.2.2 can take advantage of this feature.

For example, you can write an applet for version 2.2.2 of the Java Card platform that is capable of handling security on one channel, while another applet attempts to access user personal information on another channel, using security information on the first. By following this design, it is possible to access information owned by a different applet without having to deselect the currently selected applet that is handling session information. Thus, you avoid losing your session-specific security data, which is usually stored in `CLEAR_ON_DESELECT` RAM memory.

On dual interface cards, each interface itself can handle up to twenty independent logical channels. Each interface has its separate pool of logical channels: channels sharing the same number on two distinct interfaces will be treated as two independent, separate logical channels. Therefore, a dual concurrent interface card could, in theory, support up to forty concurrent logical channels, twenty per each interface. Channel management commands can only affect the operation logical channels in the interface where these commands were issued.

For more information on logical channels, their implementation and logical channel terminology, see the *Runtime Environment Specification for the Java Card Platform, Version 2.2.2*.

Applets and Logical Channels

In version 2.2.2 of the Java Card platform, you can work with applets that are aware of multiple channels and applets that are not aware of multiple channels.

The logical channel implementation in version 2.2.2 of the Java Card platform preserves backward compatibility with applets written for the Java Card platform version 2.1. It also allows you the option of writing your applets to use the logical channel feature or of writing the applets to work independently on any channel without using the logical channels at all.

Non-MultiSelectable Applets

In version 2.2.2 of the Java Card platform, you have the option of writing applets that can operate in a multiple channel environment, or you can write applets that do not take advantage of this feature. Applets written for the Java Card platform that do not take advantage of the multiple channel environment are similar to applets written for the version 2.1 Java Card specification. An applet written for the Java Card platform that is not designed to be aware of multiple channels cannot be selected more than once nor can any other applet inside the package be selected concurrently on a different channel.

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

Interoperability

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

Understanding the MultiSelectable Interface

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

If an applet that does not implement `MultiSelectable` is selected more than once on different channels, or selected concurrently with applets in the same package, an error is returned to the terminal.

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

The `MultiSelectable` interface contains a `select` and a `deselect` method to manage multiselectable applets.

Selection for MultiSelectable Applets

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

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

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

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

Deselection for MultiSelectable Applets

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

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

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

This method can be called as the result of a `MANAGE CHANNEL CLOSE` or `SELECT FILE` APDU command. If the applet still remains active on a different channel, the `appInstStillSelected` parameter is passed as `true`. Note that if the `MultiSelectable.deselect(boolean)` method is called, it means that either an instance of this applet or another applet from the same package remains active on another channel, so `CLEAR_ON_DESELECT` transients are not cleared. Only when the last applet instance from the entire package is deselected does a call to `Applet.deselect()` result, resulting in the erasure of `CLEAR_ON_DESELECT` transients.

Writing Applets For Concurrent Logical Channels

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

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

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

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

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

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

```
public boolean select(boolean appInstAlreadySelected) {

    // Implement the logic to control applet selection
    // during a multiselection situation
    ...
}

public void deselect(boolean appInstStillSelected) {

    // Implement the logic to control applet deselection
    // during a multiselection situation
    ...
}
```

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

MultiSelectable Applet Example

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

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

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

Also, assume that the applet has the following requirements:

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

Assume that the following methods are responsible for clearing and setting the data:

```
// dataType parameter as above
final static byte DATA_PRIVATE       = (byte)01;
final static byte DATA_PACKAGE       = (byte)02;
...

public void initData(byte[] dataArray, byte dataType) {
...
}

public void clearData(byte[] dataArray) {
...
}
```

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

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

```
public boolean select() {

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

    return true;

}
```

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

```
public boolean select(boolean appInstAlreadySelected) {

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

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

```
public void deselect(boolean appInstStillSelected) {

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

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

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

}
```

Handling Channel Information on APDU Commands

APDU commands follow the ISO 7816-4:2005 specifications to encode logical channel information. The CLA byte encodes logical channel information. The CLA byte encoding is divided into two spaces: interindustry, used by all ISO 7816-4:2005-defined commands, and the proprietary space, used by Java Card technology to encode application- specific commands.

The CLA byte encoding is divided into two classes: Type 4 commands, which encode legacy ISO 7816-4 logical channel information; and Type 16 commands, which are defined by the ISO 7816-4:2005 specification to encode information for additional 16 logical channels in the card. Type 4 logical channels occupy the range of [0..3], while Type 16 logical channels go in the range of [4..19], that is, the value encoded in the CLA byte plus four, as it is used in SELECT, MANAGE CHANNEL and other proprietary or ISO commands.

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

The CLA byte encoding follows the following rules.

Interindustry Space

CLA	Remarks
0x0X	Type 4, last or only command in chain
0x1X	Type 4, not last command in chain (paired with 0x0X)
0x2X	RFU (will throw exception)
0x3X	RFU (will throw exception)
0x4X	Type 16, no SM, last or only command in chain
0x5X	Type 16, no SM, not last command in chain (paired with 0x4X)
0x6X	Type 16, SM, last or only command in chain
0x7X	Type 16, SM, not last command in chain (paired with 0x07X)

The encoding details are as follows.

Type 4:

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

Type 16:

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

Notation:

x = Command Chaining bit

0 = last or only command

1 = command chaining

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

z = Logical channel indicator

Type 4 supports logical channels [0..3]

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

Proprietary Java Card Technology Space

CLA	Remarks
0x8X	Type 4, last or only command in chain
0x9X	Type 4, not last command in chain (paired with 0x8X)
0xAX	Type 4, last or only command in chain
0xBX	Type 4, not last command in chain (paired with 0xAX)
0xCX	Type 16, no SM, last or only command in chain
0xDX	Type 16, no SM, not last command in chain (paired with 0xCX)
0xEX	Type 16, SM, last or only command in chain
0xFX	Type 16, SM, not last command in chain (paired with 0xEX)

The encoding details are as follows.

Type 4:

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

Type 16:

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

All applets willing to use the logical channel capabilities must comply with the ISO 7816-4:2005 CLA byte encoding specification, and choose APDU commands as defined in the proprietary space.

The `X` nibble is responsible for logical channels and secure message encoding. Only the two least significant bits of the nibble are used for channel encoding, which ranges from `0` to `3`. When an `APDU` command is received, the card processes it and determines whether the command has logical channel information encoding. If logical channel information is encoded, then the card sends the `APDU` command to the respective channel. All other `APDU` commands are forwarded to the card's basic channel (0). For example, the command 0xB1 forwards the command to the card's basic channel (0), because the CLA byte with the nibble 0xBX does not contain logical channel information.

This also means that all applets willing to use the logical channel capabilities must comply with the ISO 7816-4 CLA byte encoding specification, and choose APDU commands accordingly.

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

To assist applet developers in correctly identifying proprietary and interindustry commands, the following API call can be used. This call returns true if the CLA byte encoding corresponds to the interindustry space, or false if it corresponds to the proprietary space.

```
...

// Applet's process method
public void process(APDU apdu) {

    byte[] buffer = apdu.getBuffer();

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

Writing ISO 7816-4:2005 Compliant Applets

If your applets must be compliant with the ISO 7816-4:2005 specification, then you must track the applet security state on each channel where it is active. Additionally, in the case of multiselectable applets, you must copy the state (including its security configuration) when you perform MANAGE CHANNEL commands from a channel other than the basic channel.

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

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

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

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

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

ISO 7816-4:2005 Compliant Applet Example

In case of a `MANAGE CHANNEL` command from a non-zero channel to another non-zero channel, the ISO 7816-4 specification requires that the security state from the applet selected in the origin channel be copied into the new channel. In the example presented in this section, assume that the state information is stored in the array `appState` inside the applet:

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

You can use the `APDU.getCLAChannel()` and `JCSystem.getAssignedChannel()` methods to identify if the applet selection case corresponds to an ISO 7816-4 case where the security state needs to be copied. Note that if such an event occurs, it will also be a multiselection situation, where the applet is also selected on the newly opened channel.

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

```
public boolean select(boolean appInstAlreadySelected) {
    ...
    // Obtain logical channels information
    // This call returns the channel where
    // the command was issued
    byte origChannel = APDU.getCLAChannel();

    // This call returns the channel where the applet is being
    // selected
    byte targetChannel = JCSystem.getAssignedChannel();

    if (origChannel == targetChannel) {
        // This is a SELECT FILE command.
        // Do processing here.
        ...
    }

    if (origChannel == 0) {
        // This is a MANAGE CHANNEL command from channel 0.
        // ISO 7816-4 state sharing case does not
        // apply here.
        // Do processing here.
        ...
    } else {
        // Since origChannel != 0, the special
        // ISO 7816-4 case applies.
        // Copy security state from origin channel
        // to assigned logical channel.
        appState[targetChannel] = appState[origChannel];

        // Do further processing here
        ...
        }
        ...
}
```

Please refer to the *Application Programming Interface for the Java Card Platform, Version 2.2.2* for more information about the API methods described above.

Applet Firewall Operation Requirements

To ensure proper operation and protection, a number of applet firewall checks have been added to the virtual machine for the Java Card virtual machine (Java Card VM) regarding security checks on method invocations.

Applets that implement `MultiSelectable` are designed to handle calls to Shareable objects across packages when several applets are active on different logical channels. In contrast, an applet written for version 2.1 of the Java Card platform, or an applet written for version 2.2.2 that does not implement `MultiSelectable`, has exclusive control over any changes to its internal objects or data when it is selected. Only when the non-multiselectable applet is in a deselected state can other applets modify its internal data structures. Therefore, if an applet is non-multiselectable, no calls to its Shareable objects can be made when it is selected.

Working with Non-MultiSelectable Applets

Applets written for version 2.2.2 of the Java Card platform do not have to implement the `MultiSelectable` interface. In this case, the applet assumes that it is uniquely selected and its owned objects will not be modified via Shareable interface objects while it is selected. The limitations are imposed when you interact with applets that do not implement `MultiSelectable`:

- It is not possible to select more than one applet simultaneously from a package if any of the applets you want to select does not implement the `MultiSelectable` interface.
- It is not possible to invoke methods of a Shareable object belonging to a non-multiselectable applet when an applet, belonging to the same group context, is active.

ISO 7816-4:2005 Specific `APDU` Commands for Logical Channel Management

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

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

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

- Origin logical channel values are encoded in the two least significant bits of the CLA byte.

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

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

- Managing logical channels
- Deselecting applets
- Selecting applets

MANAGE CHANNEL OPEN

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

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

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

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

:

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

This command produces the following results:

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

This command returns an error under the following circumstances:

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

MANAGE CHANNEL CLOSE

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

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

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

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

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

This command returns an error in the following circumstances:

- Channel encoded in R is the basic channel.

- Channel encoded in Q is not open.

It returns a warning if channel R is not open.

SELECT FILE

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

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

This SELECT FILE command selects an applet on channel R:

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

This command produces the following results:

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

This command returns an error in the following circumstances:

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

CHAPTER 3

Working with APDU I/O

APDU I/O is a library included with the Java Card development kit. This library is used by many Java Card development kit components, such as `apdutool`, Java Card platform Workstation Development Environment (Java Card WDE), and the RMI client framework.

This library can also be used by developers to develop Java Card client applications and Java Card platform simulators. It provides the means to exchange APDUs by using the T=0 protocol over TLP224, by using T=1, and by using the PC/SC API. (However, note that PC/SC is unsupported and may not work on all platforms with all card readers).

The library is located in the Java Archive (JAR) file `apduio.jar`.

The APDU I/O API

All publicly available APDU I/O client classes are located in the package `com.sun.javacard.apduio`. The following describes the APDU I/O API.

Javadoc tool files for the APDU I/O API are located in this bundle in HTML format at `java_card_kit-2_2_2/doc/en/dev-notes/html/apduiojavadocs` and a compilation of them in PDF format at `java_card_kit-2_2_2/doc/en/dev-notes/pdf/apdiojavadocs.pdf`.

APDU I/O Classes and Interfaces

The APDU I/O classes and interfaces are described in this section.

- `class Apdu`

Represents a pair of APDUs (both C-APDU and R-APDU). Contains various helper methods to access APDU contents and constants providing standard offsets within the APDU.

- `interface CadClientInterface`

Represents an interface from the client to the card reader or a simulator. Includes methods for powering up, powering down and exchanging APDUs.

  - `void exchangeApdu(Apdu apdu)`

  Exchanges a single APDU with the card. Note that the APDU object contains both incoming and outgoing APDUs.

  - `public byte[] powerUp()`

  Powers up the card and returns ATR (Answer-To-Reset) bytes.

  - `void powerDown(boolean disconnect)`

  Powers down the card. The parameter, applicable only to communications with a simulator, means "close the socket". Normally, it is `true` for contacted connection, `false` for contactless. See "Two-interface Card Simulation" on page 25 for more details.

  - `void powerDown()`

  Equivalent to `powerDown(true)`.

- `abstract class CadDevice`

Factory and a base class for all `CadClientInterface` implementations included with the APDU I/O library. Includes constants for the T=0, T=1 and PC/SC (unsupported) clients.

  The factory method `static CadClientInterface getCadClientInstance(byte protocolType, InputStream in, OutputStream out)`, returns a new instance of `CadClientInterface`. The in and out streams correspond to a socket connection to a simulator. Protocol type can be one of:

  - `CadDevice.PROTOCOL_T0`
  - `CadDevice.PROTOCOL_T1`
  - `CadDevice.PROTOCOL_PCSC`

  The parameters, `InputStream` and `OutputStream`, are not used for PC/SC (unsupported).

Exceptions

Various exceptions may be thrown in case of system malfunction or protocol violations. In all cases, their `toString()` method returns the cause of failure. In addition, `java.io.IOException` may be thrown at any time if the underlying socket connection is terminated or could not be established.

- `CadTransportException` extends `Exception`
- `T1Exception` extends `CadTransportException`
- `TLP224Exception` extends `CadTransportException`

Two-interface Card Simulation

To simulate dual-interface cards with the C-language Java Card RE and Java Card WDE, the following model is used:

- The simulator (`cref` or Java Card WDE) listens for communication on two TCP sockets: (*n*) and (*n*+1), where *n* is the default (9025) or the socket number given in the command line.
- The client creates two instances of the `CadClientInterface`, with protocols T=1 on both. One of these instances communicates on the port (*n*), while the other communicates on the port (*n*+1).
- Each of these client interfaces needs to issue the `powerUp` command before being able to exchange APDUs.
- Issuing the `powerDown` command on the contactless interface closes all contactless logical channels. After this, the contacted interface is still available to exchange APDUs. The client also may issue `powerUp` on a contactless interface again and continue exchanging APDUs on the contactless interface too.
- Issuing the `powerDown` command on the contacted interface closes all channels and causes the simulator (`cref` or Java Card WDE) to exit. That is, any activity after powering down the contacted interface requires restarting the simulator and reestablishing connections between the client and the simulator.
- At most, one socket can be processing an APDU at any time. The client may send the next APDU only after the response of the previous APDU is received. This means, behavior of the client+simulator still remains deterministic and reproducible.
- If you have a source release of the Java Card development kit, you can see a sample implementation of such a dual-interface client in the file `ReaderWriter.java` inside the `apdutool` source tree.

Examples of Use

The following sections give examples of how to use the APDU I/O API.

To Connect To a Simulator

To establish a connection to a simulator (such as `cref` of Java Card WDE), use the following code.

```
CadClientInterface cad;
Socket sock;
sock = new Socket("localhost", 9025);
InputStream is = sock.getInputStream();
OutputStream os = sock.getOutputStream();
cad=CadDevice.getCadClientInstance(CadDevice.PROTOCOL_T0, is, os);
```

This code establishes a T=0 connection to a simulator listening to port `9025` on `localhost`. To open a T=1 connection instead, in the last line replace `PROTOCOL_T0` with `PROTOCOL_T1`.

Note: for dual-interface simulation simply open two T=1 connections on ports (*n*) and (*n*+1), as described in "Two-interface Card Simulation" on page 25.

To Establish a T=0 Connection To a Card

To establish a T=0 connection to a card inserted in a TLP224 card reader, which is connected to a serial port, use the following code.

```
String port = "com1";  // serial port's name
CommPortIdentifier portId = CommPortIdentifier.getPortIdentifier(port);
String appname = "Name of your application";
int timeout =  30000;
CommPort commPort = portId.open(appname, timeout);
InputStream is = commPort.getInputStream();
OutputStream os = commPort.getOutputStream();
cad=CadDevice.getCadClientInstance(CadDevice.PROTOCOL_T0, is, os);
```

Note: for this code to work, you need a TLP224-compatible card reader, which is not widely available. You will also need the `javax.comm` library installed on your machine. See the *Development Kit User's Guide for the Java Card Platform, Version 2.2.2* for details on how to obtain this library.

To Establish a Connection To a PC/SC-Compatible Card Reader

To establish a connection to the default PC/SC-compatible card reader (unsupported) installed on the machine, use the following code.

```
cad=CadDevice.getCadClientInstance(CadDevice.PROTOCOL_PCSC, null,
null);
```

To Power Up And Power Down the Card

To power up the card, use the following code.

```
cad.powerUp();
```

To power down the card and close the socket connection (for simulators only), use either of the following code lines.

```
cad.powerDown(true);
```

or

```
cad.powerDown();
```

To power down, but leave the socket open, use the following code. If the simulator continues to run (which is true if this is contactless interface of the C-language Java Card RE or Java Card WDE), you can issue `powerUp()` on this card again and continue exchanging APDUs.

```
cad.powerDown(false);
```

The dual-interface C-language Java Card RE is implemented in such a way that once the client establishes connection to a port, the next command must be `powerUp` on that port.

For example, the following sequence is valid:

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

However, the following sequence is not valid:

1. **Connect on "contacted" port.**
2. **Connect on "contactless" port.**
3. **Send `powerUp` to any port.**

To Exchange APDUs

To exchange APDUs, first create a new APDU object using the following code:

```
Apdu apdu = new Apdu();
```

Copy the header `(CLA, INS, P1, P2)` of the APDU to be sent into the `apdu.command` field.

Set the data to be sent and the `Lc` using the following code:

```
apdu.setDataIn(dataIn, Lc);
```

where the array `dataIn` contains the C-APDU data, and the `Lc` contains the data length.

Set the number of bytes expected into the `apdu.Le` field.

Exchange the APDU with a card or simulator using the following code:

```
cad.exchangeApdu(apdu);
```

After the exchange, `apdu.Le` contains the number of bytes received from the card or simulator, `apdu.dataOut` contains the data received, and `apdu.sw1sw2` contains the SW1 and SW2 status bytes.

These fields can be accessed through the corresponding `get` methods.

To Print the APDU

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

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

CHAPTER 4

Developing RMI Applications for the Java Card Platform

This chapter describes how to write RMI applications for the Java Card platform. In this release, you can run and debug Java Card remote method invocation (Java Card RMI) applications in the C language Java Card RE and the Java Card platform Workstation Development Environment (Java Card WDE).

Developing an RMI Applet for the Java Card Platform

Following are the main steps for developing an RMI applet for the Java Card platform:

1. **Define remote interfaces**
2. **Develop classes implementing the remote interfaces**
3. **Develop the `main` class for the applet**

   For a simple applet, the main class of the applet can also be the class implementing the remote interface.

Generating Stubs

The Java Card RMI Client framework requires stubs only when the `remote_ref_with_class` format is used for passing remote references. These stubs of remote classes of applets must be pr-generated and available on the client. When the `remote_ref_with_interfaces` format is used, stubs are not necessary.

In this example, Sun Microsystems' standard RMI Compiler (`rmic`) is used to generate these stubs.

Following is the command to run the `rmic`:

`rmic -v1.2 -classpath` *path* `-d` *output_dir class_name*

where:

path includes the path to the remote class of your sample applet and to the file `javacardframework.jar`.

output_dir is the directory in which to place the resulting stubs

class_name is the name of the remote class

The `-v1.2` flag is required by the RMI client framework for the Java Card platform.

The `rmic` must be called for each remote class in your applet.

Note – You need to generate stubs only for remote classes that list a remote interface in their `implements` clause.

The file `javacardframework.jar` is provided in version 2.2.2 of the Java Card development kit. This JAR file contains compiled implementations of packages `javacard.framework`, `javacard.framework.service`, and `javacard.security`. Classes in these packages might be referenced by Java Card RMI applets and thus might be needed by the `rmic` to generate stubs.

Running a Java Card RMI Applet

The server part (the Java Card RMI-enabled applet) can be run on both the C-language Java Card RE and Java Card WDE.

To run the applet on the C-language Java Card RE, the standard procedures apply: the applet must be installed first, using the installer applet. After the applet is installed, the EEPROM state can be saved and used to run the C-language Java Card RE against the Java Card RMI client.

The simplest way to run a Java Card RMI-enabled applet on the Java Card WDE is to add it to the WDE configuration file on the first line. This uses the fact that the Java Card WDE automatically installs the first applet on "power up." The Java Card WDE is a very convenient environment to debug Java Card RMI applets. Of course, all of the standard limitations (such as absence of firewall support) apply.

Running the Java Card RMI Client Program

The client program can be developed and compiled using `javac` or your favorite IDE. To compile the client, the remote interfaces for your applet must be present in your `classpath`.

Running the client program has the following requirements.

- The client framework file `jcrmiclientframework.jar` is present in the `classpath`. This file contains all the client framework and necessary classes from the card framework.
- The file `jccclient.properties` is present in one of the directories specified in the `classpath`.
- The remote interfaces and stubs for your applet are present in the `classpath`. For a sample command line to run a client program, refer to the file `rmidemo` or `rmidemo.bat` in this directory.
- The `jccclient.properties` file supplied in the `samples/src/demo` directory. This file specifies parameters for `com.sun.javacard.clientlib.APDUIOCardAccessor`. To be accessible at runtime, this file must be located in one of the directories listed in the `classpath`. This parameter connection specified in the file can be configured to be TCP, serial or PC/SC (unsupported). The protocol being used can be T0 or T1.

Basic Example

The basic example is the Java Card platform equivalent of “Hello World,” which is a program that manages a counter remotely, and is able to decrement, increment, and return the value of the counter.

Main Program

As for any Java Card RMI program, the first step is to define the interface to be used as contract between the server (the Java Card technology-based application) and its clients (the terminal applications):

```
package examples.purse ;
import java.rmi.* ;
import javacard.framework.* ;
public interface Purse extends Remote {
  public static final short MAX_AMOUNT = 400 ;
  public static final short REQUEST_FAILED = 0x0102 ;
  public short debit(short amount) throws RemoteException, UserException;
  public short credit(short amount) throws RemoteException,
      UserException ;
  public short getBalance() throws RemoteException, UserException ;
}
```

This is a typical Java Card RMI interface in the following ways:

- The interface type extends the `java.rmi.Remote` interface. This interface is a tagging interface that identifies the interface as defining a remotely accessible object.
- Every method in the interface must be declared as throwing a `RemoteException` or one of its superclasses (`IOException` or `Exception`). This exception is required to encapsulate all the communication problems that might occur during a remote invocation of the method. In addition, the `credit`, `debit`, and `getBalance` methods also throw the `UserException` to indicate application-specific errors.
- The interface can also define values for constants that might be used in the communication between the client and the server. The `Purse` interface defines a constant `MAX_AMOUNT` that represents the maximum allowed value for the transaction amount parameter. It also defines a reason code `REQUEST_FAILED` for the `UserException` qualifier.

Implement a Remote Interface

The next step provides an implementation for this interface. This implementation runs on a Java Card platform, and it therefore needs to use only features that are supported by a Java Card platform:

```
package examples.purse ;
import javacard.framework.* ;
import javacard.framework.service.* ;
import java.rmi.* ;
public class PurseImpl extends CardRemoteObject implements Purse
{
  private short balance ;
  PurseImpl()
  {
    super() ;
    balance = 0 ;
  }
  public short debit(short amount) throws RemoteException, UserException
  {
    if (( amount < 0 )||( amount > MAX_AMOUNT ))
      UserException.throwIt(REQUEST_FAILED) ;
    balance -= amount ;
    return balance ;
  }
  public short credit(short amount) throws RemoteException, UserException
  {
    if (( amount < 0 )||( balance < amount ))
      UserException.throwIt(REQUEST_FAILED) ;
    balance -= amount ;
    return balance ;
  }
  public short getBalance() throws RemoteException, UserException
  {
    return balance ;
  }
}
```

Here, the remote interface is the `Purse` interface, which declares the remotely accessible methods. By implementing this interface, the class establishes a contract between itself and the compiler, by which the class promises that it will provide method bodies for all the methods declared in the interface:

```
public class PurseImpl extends CardRemoteObject implements Purse
```

The class also extends the `javacard.framework.service.CardRemoteObject` class. This class provides our class with basic support for remote objects, and in particular the ability to export or unexport an object.

Define the Constructor for the Remote Object

The constructor for a remote class provides the same functionality as the constructor of a non-remote class: it initializes the variables of each newly created instance of the class.

In addition, the remote object instance will need to be exported. Exporting a remote object makes it available to accept incoming remote method requests. By extending `CardRemoteObject`, a class guarantees that its instances are exported automatically upon creation on the card.

If a remote object does not extend `CardRemoteObject` (directly or indirectly), you must explicitly export the remote object by calling the `CardRemoteObject.export` method in the constructor of your class (or in any appropriate initialization method). Of course, this class must still implement a remote interface.

To review:

The implementation class for a remote object needs to do the following:

- Implement a remote interface
- Export the object so that it can accept incoming remote method calls

Provide an Implementation for Each Remote Method

The implementation class for a remote object contains the code that implements each of the remote methods specified in the remote interface. For example, here is the implementation of the method that debits the purse:

```
public short debit(short amount) throws RemoteException, UserException

    if (( amount < 0 )||( balance < amount )
      UserException.throwIt(REQUEST_FAILED) ;
    balance -= amount ;
    return balance ;
  }
```

An operation is only allowed if the value of its parameter is compatible with the current state of the purse object. In this particular case, the application only checks that the amounts handled are positive and that the balance of the purse always remains positive.

In Java Card RMI, the arguments to and return values from remote methods are restricted. The main reason for this limitation is that the Java Card platform does not support object serialization. Following are the rules for the Java Card platform:

- The arguments to remote methods can be of any supported integral type (such as `boolean`, `byte`, `short` and `int`), or any single-dimensional arrays of these integral types.

Note – The `int` type is optionally supported on the Java Card platform, so applications that use this type might not run on all platforms.

- The return value from a remote method can be any type supported as arguments, as well as any remote interface type. The method can also return `void`.

On the other hand, object passing in Java Card RMI follows the normal RMI rules:

- By default, non-remote objects are passed by copy, which means that all data members of an object are copied, except those marked `static` or `transient`. In the case of the Java Card platform, this rule is trivial to apply, because the only objects concerned are arrays of integral types.
- Remote objects are passed by reference. In the case of the Java Card platform, remote objects can only be passed as return values. A reference to a remote object is actually a reference to a stub, which is a client-side proxy for the remote objects. Stubs are needed only when the format `remote_ref_with_class` is used for passing remote references. When another format, such as `remote_ref_with_interfaces`, is used, stubs are not necessary. Stubs are described in "Generate the Stubs" on page 41.

Note – Even though the semantics of the Java Card platform transient arrays are somewhat similar to transient fields in the Java programming language, different rules apply. Java Card platform contents are copied in Java Card RMI and passed by value when they are returned from a remote method.

A class can define methods not specified in a remote interface, but they can only be invoked on-card within the Java Card VM and cannot be invoked remotely.

Building an Applet

In version 2.2.2 of the Java Card platform (as in version 2.1), all applications must include a class that inherits from `javacard.framework.Applet`, which will provide an interface with the outside world. This also applies to applications that are based on remote objects, for two main reasons:

- The remote objects must be instantiated and initialized, which can be done in an applet's `install` method.
- The remote objects must communicate with the outside world, which can be done in an applet's `process` method.

For conversion, an applet should be assigned with an AID known on the client side, `0x00;0x01:0x02:0x03:0x04:0x05:0x06:0x07:0x08`, since this AID is used in the client program.

Following is the basic code for such an applet:

```
package examples.purse ;
import javacard.framework.* ;
import javacard.framework.service.* ;
import java.rmi.*;
public class PurseApplet extends Applet
{
  private Dispatcher dispatcher ;
  private PurseApplet()
  {
    // Allocates an RMI service and sets for the Java Card platform
    // the initial reference
    RemoteService rmi = new RMIService( new PurseImpl() ) ;
    // Allocates a dispatcher for the remote service
    dispatcher = new Dispatcher((short)1) ;
    dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND) ;
  }
  public static void install(byte[] buffer, short offset, byte length)
  {
    // Allocates and registers the applet
    (new PurseApplet()).register() ;
  }
  public void process(APDU apdu)
  {
    dispatcher.process(apdu) ;
  }
}
```

Preparing and Registering the Remote Object

The `PurseApplet` constructor contains the initialization code for the remote object. First, a `javacard.framework.service.RMIService` object must be allocated. This service is an object that knows how to handle all the incoming `APDU` commands related to the Java Card RMI protocol. The service must be initialized to allow remote methods on an instance of the `PurseImpl` class. A new instance of `PurseImpl` is created, and is specified as the initial reference parameter to the `RMIService` constructor as shown in the following code snippet. The initial reference is the reference that is made public by an applet to all its clients. It is used as a bootstrap for a client session, and is similar to that registered by a Java RMI server to the Java Card RMI registry.

```
RemoteService rmi = new RMIService( new PurseImpl() ) ;
```

Then, a dispatcher is created and initialized. A dispatcher is the glue among several services. In this example, the initialization is quite simple, because there is a single service to initialize:

```
dispatcher = new Dispatcher((short)1) ;

dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND) ;
```

Finally, the applet must register itself to the Java Card RE to be made selectable. This is done in the `install` method, where the applet constructor is invoked and immediately registered:

```
(new PurseApplet()).register() ;
```

Processing the Incoming Commands

The processing of the incoming commands is entirely delegated to the Java Card RMI service, which knows how to handle all the incoming requests. The service also implements a default behavior for the handling of any request that it does not recognize. In Java Card RMI, the following kinds of requests that can be handled:

- Selection request, to which the service responds by sending its initial remote reference
- Method invocation request, to which the service responds by performing the actual method invocation and returning the result

To perform these actions, the service needs privileged access to some resources that are owned by the Java Card RE (in particular, privileged access is needed to perform the method invocation). The applet delegates processing to the Java Card RMI service from its process method as follows:

```
dispatcher.process(apdu) ;
```

Writing a Client

The client application runs on a terminal supporting a Java Virtual Machine[1] environment such as Java 2 Platform, Standard Edition (J2SE™ platform) or Java 2 Platform, Micro Edition (J2ME™ platform).

The `PurseClient` application interacts with the remote stub classes generated by a stub generation tool and the Java Card platform-specific information managed by the Java Card platform client-side framework located in packages `com.sun.javacard.clientlib` and `com.sun.javacard.rmiclientlib`.

The following example uses standard Java RMIC compiler-generated client-side stubs. The client application as well as the Java Card client-side framework rely on the APDU I/O library for managing and communicating with the card reader and the card on which the Java Card applet `PurseApplet` resides. This makes the client application very portable on J2SE platforms.

The following example shows a very simple `PurseClient` application that is the client application of the Java Card technology-based program `PurseApplet`:

1. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java platform.

```
import examples.purse.* ;
import javacard.framework.UserException ;

  public class PurseClient extends java.lang.Object {

public static void main(java.lang.String[] argv) {
// arg[0] contains the debit amount
short debitAmount = (short) Integer.parseInt(argv[0]) ;

          CardAccessor ca = null;
try {
               // open and powerup the card
               ca = new ApduIOCardAccessor();
// create an RMI connector instance for the Java Card
platform
               JCRMIConnect jcRMI = new JCRMIConnect(ca);

byte[] appAID = new byte[] {0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08};
               // select the Java Card applet
                    jcRMI.selectApplet( RMI_DEMO_AID,
JCRMIConnect.REF_WITH_CLASS_NAME );

or

jcRMI.selectApplet( RMI_DEMO_AID, JCRMIConnect.REF_WITH_INTERFACE_NAMES );

// obtain the initial reference to the Purse interface
Purse myPurse = (Purse) jcRMI.getInitialReference() ;
// debit the requested amount

try {
short balance = myPurse.debit ( debitAmount ) ;
}catch ( UserException jce ) {
short reasonCode = jce.getReason() ;
// process UserException reason information
}
// display the balance to user

}catch (Exception e) {
e.printStackTrace() ;
} finally {

try {
if(ca!=null){
                        ca.closeCard();
                   }
}catch (Exception e) {
e.printStackTrace() ;
}
}
}
}
```

Initializing and Shutting Down the Card Connection

The client application must open the connection to the card and close it at the end.

Note – ApduIOCardAccessor takes its settings from the file `jcclient.properties`. When one of the RMI related demos runs, its script modifies the CLASSPATH to include this file, which is located at `java_card_kit-2_2_2/samples/src_client` in the binary release bundle for Solaris™ or Linux platforms and at `java_card_kit-2_2_2\samples\src_client` on the Windows platform.

The following code shows opening and closing the connection using the RMI client framework:

```
CardAccessor ca = null;

// The following line initializes card connection according to
// parameters listed in the jcclient.properties file:
ca = new ApduIOCardAccessor();

...

// The following line powers down the card and closes the connection:
ca.closeCard();
```

Creating and Using a `CardAccessor` *Object*

To access the Java Card applet using remote methods, the client application must obtain an instance of the `CardAccessor` interface. The `ApduIO` class implements the `CardAccessor` interface and is included in the framework.

The `CardAccessor` interface is a platform-independent and framework-independent interface that is used by the RMI framework for the Java Card platform to communicate with the card. The `CardAccessor` object is then provided as a parameter during construction of the `JavaCardRMIConnect` class to initiate an RMI dialogue for the Java Card platform as the following code shows:

```
// create an RMI connection object for the Java Card platform

JavaCardRMIConnect jcRMI = new JavaCardRMIConnect( myCS ) ;
```

Selecting the Java Card Applet and Obtaining the Initial Reference

To invoke methods on the remote objects of the Java Card applet `PurseApplet` on the card, it must first be selected using the AID:

```
// select the Java Card applet
  byte[] appAID = new byte[] {0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08} ;
  jcRMI.selectApplet( appAID ) ;
```

Then, the client must obtain the initial reference remote object for `PurseApplet`. `JavaCardRMIConnect` returns an instance of a stub class corresponding to the `PurseImpl` class on the card which implements the `Purse` interface. The client application knows beforehand that the `PurseApplet`'s initial remote reference implements the `Purse` interface and therefore casts it appropriately:

```
// obtain the initial reference to the Purse interface

Purse myPurse = (Purse) jcRMI.getInitialReference() ;
```

Using Remote Objects in Remote Method Invocations

The client can now invoke remote methods on the initial reference object. The remote methods are declared in the `Purse` interface. The following code shows the client invoking the `debit` method. Note how an `UserException` exception thrown by the remote method is caught by the client code in a normal Java programming language style.

```
// debit the requested amount
try {
    short balance = myPurse.debit ( debitAmount ) ;
    }catch ( UserException jce ) {
    short reasonCode = jce.getReason() ;
    // process on card exception reason information
}
```

Generate the Stubs

The client-side scenario uses RMIC generated stubs for the remote classes. RMIC is the Java RMI stub compiler. For the client application `PurseClient` to execute correctly on the terminal, it needs these remote stub classes and the remote interface class files it uses to be accessible in its classpath.

The stub class `PurseImpl_Stub.class` for the `PurseImpl` class is produced by running the standard JDK1.5 RMIC compiler. For example, when in the `examples/purse` directory, enter the following commands:

Solaris and Linux platforms:

```
rmic -classpath ../..;$JC_HOME/lib/javacardframework.jar -d ../..
-v1.2 examples.purse.PurseImpl
```

Microsoft Windows platform:

```
rmic -classpath ../..;%JC_HOME%/lib/javacardframework.jar -d ../..
-v1.2 examples.purse.PurseImpl
```

This produces a stub class called `examples.purse.PurseImpl_Stub`.

Thus, for `PurseClient` to run correctly on the terminal, the following files must be present in the `examples/purse` directory and accessible via its classpath or from class loaders:

- `PurseImpl_Stub.class`
- `Purse.class`

Card Terminal Interaction

When a Java Card technology-enabled smart card is powered up, the card sends an ATR (Answer to Reset) to the terminal. The Card Accessor returns the value of the ATR to the client program.

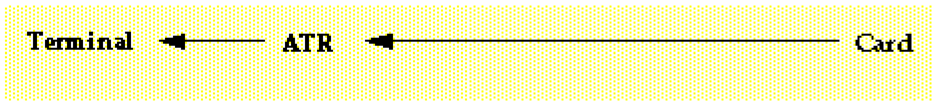

FIGURE 4-1 Smart Card Sends an ATR to the Terminal

When the `PurseClient` application calls the `selectApplet` method of `JavaCardRMIConnect`, it sends a `SELECT APDU` command to the card via the `CardAccessor` object. This results in a File Control Information (FCI) APDU response from the `RMIService` instance of `PurseApplet` on the card in a TLV (Tag Length Value) format that includes the initial reference remote object information, which FIGURE 4-2 illustrates.

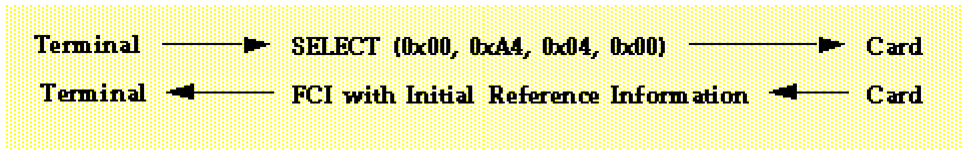
Terminal
SELECT (0x00, 0xA4, 0x04, 0x00)
Card
Terminal
FCI with Initial Reference Information
Card

FIGURE 4-2 Terminal Sends a `SELECT` Command to the Smart Card, which Returns FCI

Later, when the `PurseClient` application calls the `debit` method of the remote interface `Purse`, the `PurseImpl_Stub` object sends an invoke command to the card via the `CardAccessor` object, identifying the remote object reference, interface, method, and parameter data for method invocation. The `RMIService` instance of `PurseApplet` unmarshalls this information and invokes the `debit` method of the `PurseImpl` instance, and returns the return value in the response `APDU`, which FIGURE 4-3 illustrates.

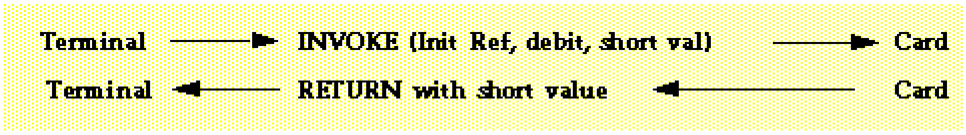

FIGURE 4-3 Terminal Sends an `INVOKE` Command to the Smart Card, Which Returns a Value

Adding Security

This first example is extremely simple and is not realistic. In particular, it does not include any kind of security. Users are not authenticated and no transport security is provided. Of course, every smart card that implements the Java Card platform includes such security mechanisms, because they are central to Java Card technology.

The following section describes how to add security support to the `Purse` example.

The `Purse` interface in the package `examples.securepurse` is similar to the `Purse` interface in the previous code sample. In addition, it might include reason codes for exceptions to report security violations to the terminal. Replace it with `examples.securepurse`. The interface does not include any implementation, which means that, in particular, it does not include any support for security.

The applet keeps its original organization but it also includes additional code that is dedicated to the management of security.

```
package examples.securepurse ;
import javacard.framework.* ;
import javacard.framework.service.* ;
import java.rmi.* ;
public class SecurePurseImpl implements Purse
{
  private short balance ;
  private SecurityService security ;
  SecurePurseImpl(SecurityService security)
  {
    this.security = security ;
  }

public short debit(short amount) throws RemoteException, UserException
  {
  if
  ((!security.isCommandSecure(SecurityService.PROPERTY_INPUT_INTEGRITY))
  ||
  (!security.isAuthenticated(SecurityService.PRINCIPAL_CARDHOLDER)))
    UserException.throwIt(REQUEST_FAILED) ;
    if (( amount < 0 )|| ( balance < amount ))
      UserException.throwIt(REQUEST_FAILED) ;
    balance -= amount ;
    return balance ;
  }

public short credit(short amount) throws RemoteException, UserException
  {
    if
    ((!security.isCommandSecure(SecurityService.PROPERTY_INPUT_INTEGRITY))
    ||
    (!security.isAuthenticated(SecurityService.PRINCIPAL_APP_PROVIDER)))
      UserException.throwIt(REQUEST_FAILED) ;
    if (( amount < 0 )||( amount > MAX_AMOUNT ))
      UserException.throwIt(REQUEST_FAILED) ;
    balance += amount ;
    return balance ;
  }

public short getBalance() throws RemoteException, UserException
  {
    if ((!security. isAuthenticated(SecurityService.PRINCIPAL_CARDHOLDER))
    &&
    (!security.isAuthenticated(SecurityService.PRINCIPAL_APP_PROVIDER)))
      UserException.throwIt(REQUEST_FAILED) ;
    return balance ;
  }
}
```

Initialize a Security Service

In this example, basic security services (principal identification and authentication, secure communication channel) are provided by an object that implements the `SecurityService` interface. Because a generic remote object must not be dependent on a particular kind of security service, it must take a reference to this object as a parameter to its constructor. This is exactly what happens here, where the reference to the object is stored in a dedicated private field:

```
private SecurityService security ;
```

The `SecurityService` interface is part of the extended application development framework and offers an API that can then be used to check on the current security status.

Use the Service to Check the Current Security Status

In the example, this following required security behaviors for the applet are assumed:

- The `debit` method is authorized only if it is sent through a secure channel that ensures at least the integrity of input data, and if the cardholder is successfully authenticated.
- The `credit` method is authorized only if it is sent through a secure channel that ensures at least the integrity of input data, and if the application issuer is successfully authenticated.
- The `getBalance` method is authorized only if the cardholder or the application issuer is successfully authenticated.

The `SecurityService` provides methods and constants that allow the implementation to perform such checks. For instance, following is the code for the checks on the `debit` method:

```
if
((!security.isCommandSecure(SecurityService.PROPERTY_INPUT_INTEGRITY))
  ||
  (security.isAuthenticated(SecurityService.ID_CARDHOLDER)))
    UserException.throwIt(REQUEST_FAILED) ;
```

If one of the two conditions is not satisfied, the remote object throws an exception. This exception is caught by the dispatcher and forwarded to the client.

Implementing a Security Service

The following example shows how to implement a security service.

```
package com.sun.javacard.samples.SecureRMIDemo ;
import javacard.framework.* ;
import javacard.framework.service.* ;

public class MySecurityService extends BasicService implements SecurityService
{
// list IDs of known parties...
    private static final byte[] PRINCIPAL_APP_PROVIDER_ID = {0x12, 0x34} ;
    private static final byte[] PRINCIPAL_CARDHOLDER_ID = {0x43, 0x21} ;
    private OwnerPIN provider_pin, cardholder_pin = null ;
    // and the security-related session flags
    ...
    public MySecurityService() {
        // initialize the PINs
        ...
    }
     public boolean processDataIn(APDU apdu) {
      if(selectingApplet()) {
           // reset all flags
            ...
      }
      else {
          return preprocessCommandAPDU(apdu);
      }
  }
  public boolean isCommandSecure(byte properties) throws ServiceException {
      // return the value of appropriate flag
      ....
  }
  public boolean isAuthenticated(short principal) throws ServiceException {
      // return the value of appropriate flag
      ....
  }
  private byte authenticated ;
  private boolean preprocessCommandAPDU(APDU apdu) {
      receiveInData(apdu) ;
      if(checkAndRemoveChecksum(apdu)) {
          // set DATA_INTEGRITY flag
        }
        else {
            // reset DATA_INTEGRITY flag
        }
        return false;   // other services may also preprocess the data
    }
    private boolean checkAndRemoveChecksum(APDU apdu) {
            // remove the checksum
            // return true if checksum OK, false otherwise
    }
    public boolean processCommand(APDU apdu) {
        if(isAuthenticate(apdu)) {
            receiveInData(apdu) ;
            // check PIN
            // set AUTHENTICATED flags
            return true;     //  processing of the command is finished
        }
        else {
             return false ;  // this command was addressed to another
                            // service - no processing is done
        }
    }
    public boolean processDataOut(APDU apdu) {
        // add checksum to outgoing data
        return false;  // other services may also postprocess outgoing data
    }
    private boolean isAuthenticate(APDU command) {
                // check values of CLA and INS bytes
    }
}
```

Building an Applet

The supporting applet also must undergo some significant changes, in particular regarding the initialization of the remote object:

```
package examples.securepurse ;
import javacard.framework.* ;
import javacard.framework.service.* ;
import java.rmi.* ;
import com.sun.javacard.samples.SecureRMIDemo.MySecurityService ;

public class SecurePurseApplet extends Applet
{
  Dispatcher dispatcher ;
  private SecurePurseApplet()
  {
    SecurityService sec ;
    // First get a security service
    sec = new MySecurityService() ;
    // Allocates an RMI service for the Java Card platform and
    // sets the initial reference
    RemoteService rmi = new RMIService( new SecurePurseImpl(sec) ) ;
    // Allocates and initializes a dispatcher for the remote object
    dispatcher = new Dispatcher((short)2) ;
    dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND) ;
    dispatcher.addService(sec, Dispatcher.PROCESS_INPUT_DATA) ;
  }
  public static void install(byte[] buffer, short offset, byte length)
  {
    // Allocates and registers the applet
    (new SecurePurseApplet()).register() ;
  }
  public void process(APDU apdu)
  {
    dispatcher.process(apdu) ;
  }
}
```

The security service that is used by the remote object must be initialized at some point. Here, this is done in the constructor for the `SecurePurseApplet`:

```
sec = new MySecurityService() ;
```

The initialization then goes on with the initialization of the Java Card RMI service. The only new thing here is that the remote object being allocated and set as the initial reference is now a `SecurePurseImpl`:

```
RemoteService rmi = new RMIService( new SecurePurseImpl(sec) );
```

Next, the dispatcher must be initialized. Here, it must dispatch simple Java Card RMI requests and security-related requests (such as `EXTERNAL AUTHENTICATE`). In fact, the security service handles these requests directly. First, allocate a dispatcher and inform it that it will delegate commands to two different services:

```
dispatcher = new Dispatcher((short)2);
```

Then, register services with the dispatcher. The security service is registered as a service that performs preprocessing operations on incoming commands, and the Java Card RMI service is registered as a service that processes the command requested:

```
dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND) ;

dispatcher.addService(sec, Dispatcher.PROCESS_INPUT_DATA) ;
```

The rest of the class (`install` and `process` methods) remain unchanged.

Writing a Client

The driver client application itself only changes minimally to account for the authentication and integrity needs of `SecurePurseApplet`. It must also interact with the user for identification. Hence, a subclass of `ApduIO_Card_Accessor` must be developed to provide these additional interactions and the transport filtering required.

Following is the new `SecurePurseClient` application:

```
import examples.purse.* ;
import javacard.framework.UserException ;

public class PurseClient extends java.lang.Object {

public static void main(java.lang.String[] argv) {
// arg[0] contains the debit amount
short debitAmount = (short) Integer.parseInt(argv[0]) ;
         CustomCardAccessor cca = null;
try {
             // open and powerup the card - using CustomCardAccessor
             cca = new CustomCardAccessor(new ApduIOCardAccessor());
// create an RMI connector instance for the Java Card
platform
             JCRMIConnect jcRMI = new JCRMIConnect(ca);

byte[] appAID = new byte[] {0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08};
// select the Java Card applet
                 jcRMI.selectApplet( RMI_DEMO_AID,
JCRMIConnect.REF_WITH_CLASS_NAME );

or

jcRMI.selectApplet( RMI_DEMO_AID, JCRMIConnect.REF_WITH_INTERFACE_NAMES );
// give your PIN
              if (! cca.authenticateUser( PRINCIPAL_CARDHOLDER_ID )){
                   throw new RemoteException(msg.getString("msg04"));
              }

// obtain the initial reference to the Purse interface
Purse myPurse = (Purse) jcRMI.getInitialReference() ;
// debit the requested amount

try {
short balance = myPurse.debit ( debitAmount ) ;
}catch ( UserException jce ) {
short reasonCode = jce.getReason() ;
// process UserException reason information
}
// display the balance to user

}catch (Exception e) {
e.printStackTrace() ;
} finally {
try {
                   if(ca!=null){
                        cca.closeCard();
                   }
}catch (Exception e) {
e.printStackTrace() ;
}
}
}
}
```

Note that the `CustomCardAccessor` instance is now obtained instead of `ApduIOCardAccessor`:

```
cca = new CustomCardAccessor(new ApduIOCardAccessor());
```

An extra step to authenticate with the `SecurePurseApplet` after `selectApplet` is added. This invokes a new method in `CustomCardAccessor` to interact with the card using the user's credentials:

```
if (! cca.authenticateUser( PRINCIPAL_CARDHOLDER_ID )) {
    // handle error
}
```

The rest of `SecurePurseClient` is the same as `PurseClient`.

Writing a CustomCardAccessor Class

The `SecurePurseClient` application uses a subclass of `CardAccessor` called `CustomCardAccessor` to perform user authentication functions and to sign every message sent thereafter for integrity purposes:

```
package examples.securepurseclient;

public class CustomCardAccessor extends
             ApduIOCardAccessor {
      /** Creates new CustomCardAccessor */
    public CustomCardAccessor() {
    }
  public byte[] exchangeAPDU( byte[] sendData )
throws java.io.IOException {

        byte[] macSignature = null ;
        byte[] dataWithMAC = new byte[ sendData.length + 4 ] ;

        // sign the sendData data using session key
        // sign the data in commandBuffer using the user's session key

        // add generated MAC signature to data in buffer before sending

        return super.exchangeAPDU( dataWithMAC ) ;
    }
    boolean authenticateUser( short userKey ) {
        byte[] externalAuthCommand = null ;

        // build and send the appropriate commands to the
        // applet to authenticate the user using the user Key
        // and additional info provided
        try {
          byte[] response = super.exchangeAPDU ( externalAuthCommand ) ;
            // ...
         }catch (Exception e) {
            // analyze
            return false ;
        }
        // Then compute the session key for later use
        return true; //successful authentication
    }
}
```

The `CustomCardAccessor` class introduces the `authenticateUser` method to send `APDU` commands to the `SecurePurseApplet` on the card to authenticate the user described by the `userKey` parameter and other parameters and to compute a transport key. It invokes `super.sendCommandAPDU` method to send the command without modification.

This `CustomCardAccessor` class also reimplements the `exchangeAPDU` method declared in a superclass `CardAccessor` to sign each message before it is sent out by `super.exchangeAPDU`.

CHAPTER 5

Using Extended APDU

The extended APDU feature in the Java Card Platform, v2.2.2, allows applet developers to take advantage of extended APDU functionality, as defined in the ISO 7816 specification. Extended APDU allows large amounts of data to be sent to the card, processed appropriately, and sent back to the terminal, in a more efficient way. Instead of having to re-issue multiple APDU messages to complete an operation requiring large volumes of data, and requiring the developer to code the application to keep a state across such multiple APDU commands, extended APDU allows applets to perform this function more efficiently with one large APDU exchange.

Extended APDU can be beneficial when dealing with large amounts of information. For example, applications such as signature verification, biometrics verification and image storage and retrieval could greatly benefit from this feature. Extended APDU implementations can easily be implemented if the underlying transport protocol is T=1, while applets developed for T=0 cards would need special logic and care to work correctly.

Extended APDU Nominal Cases

The ISO 7816-4:2005 specification defines an extended APDU as any APDU whose payload data, response data or expected data length exceeds the 256 byte limit. Therefore, the four traditional cases are redefined as follows:

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

- Case 4S. The legacy case 4. LC and LE are less than 256 bytes of data.
- Case 4E. The extended version of Case 4. LC or LE are greater than 256 bytes of data.

Extended APDU Format

To express extended length, the APDU format has changed. The table below summarizes the format defined by ISO 7816-4:2005 for extended length APDU. Any APDU classified as extended must follow this format.

TABLE 5-1 Extended APDU Format

Field	Description	Number of Bytes
Command Header	Class byte CLA	1
Command Header	Instruction byte INS	1
Command Header	Parameter bytes P1- P2	2
LC Field	Absent for Nc = 0. Present for Nc > 0	0, 1, or 3
Data Field	Absent if Nc = 0, present if Nc >0	Nc
LE Field	Absent for Ne = 0, present for Ne > 0	0, 1, 2 or 3
Response Data	Absent if Nr = 0, present if Nr >0	Nr (max. Ne)
Response Status	Status bytes SW1 SW2	2
	NOTATION Nc = command data length Ne = expected response data length Nr = actual response data length	

The encoding rules are defined as:

For LC:

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

For LE:

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

If LE is an extended field:

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

Extended APDU Limits

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

`javacardx.framework.ExtendedLength` Interface

Not all Java Card applets can handle extended APDUs. Legacy applets should never encounter an extended APDU in the APDU buffer. Because of this, the Java Card API has added a tagging interface, `javacardx.apdu.ExtendedLength`, to signal that the applet implementing this interface is capable of processing, receiving and replying to extended APDU commands. The Java Card RE will not deliver extended APDU commands to applets not implementing this interface (it would throw an `ISOException` with reason code `ISO7816.SW_WRONG_LENGTH` in that case), nor would it allow applets to send reply data lengths greater than 256, if such an interface is not implemented by the applet.

The APDU buffer in Java Card technology applications will reflect the structure of the extended APDU as defined in ISO. In T=1, this representation is straightforward and precise; whereas in T=0, there need to be some adaptations for some cases.

Specifically, a case 2E APDU sent over T=0 transport will not show its extended LE value in the APDU buffer. Instead, a P3 value of '00' will always be transmitted, and interpreted as 32,767, if the applet implements `ExtendedLength`, or 256 if it does not. The Java Card RE analyzes the APDU type coming into the card and determines its type based on the rules defined in the ISO 7816-3 specification. Because case 2E commands look like case 2S commands in T=0, the Java Card RE is not able to distinguish this particular case.

Extensions To `javacard.framework.APDU` Class

Because LC in cases 3E and 4E can take a large value, the parameter is sent to the card as a three-byte quantity, in the format of `00 LCh LCl` starting at `ISO7816.OFFSET_LC`. Two new API calls have been added to `javacard.framework.APDU` so that the applet developer will not be required to parse the APDU. The API calls allow the applet developer to get the value of LC and the data offset inside the APDU buffer without having to get them directly from that buffer, as was necessary before.

These two APIs allow applet developers to write applets without having to worry about parsing extended length in T=0 and T=1 implementations.

- `public short getIncomingLength()`

This API call returns the value of LC as expressed in the APDU, whether it is extended or not.

- `public short getOffsetCdata()`

This API call returns the offset where the first byte of the APDU data segment is found.

Sending and Receiving Extended APDU Commands

To write an applet that takes advantage of extended length, follow these steps:

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

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

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

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

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

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

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

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

        short recvLen = apdu.setIncomingAndreceive();
        short dataOffset = apdu.getOffsetCdata();

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

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

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

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

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

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

Index