Application Programming Notes

Java Card™ Platform, Version 2.2.1

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

October, 2003

Please

Adobe PostScript™

Contents

Preface

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

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

The *Application Programming Notes for the Java Card™ Platform, Version 2.2.1* contains tips and guidelines for the applet developer who is using the *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003) to implement applet management, multiselectable applets, logical channels and RMI for the Java Card platform. It is also for developers who are considering creating a vendor-specific framework based on version 2.2.1 of Java Card technology specifications.

Before You Read This Book

Before reading this guide, you should be 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 should also be familiar with the development tools released with version 2.2.1 of the Java Card platform. For information on these tools, see the *Development Kit User's Guide for the Java Card™ Platform, Version 2.2.1*.

How This Book Is Organized

Chapter 1 "Using the Object Deletion Mechanism, and Package and Applet Deletion" describes how you can perform object deletion, applet deletion, and package deletion on the Java Card platform.

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

Chapter 3 "Developing RMI Applications for the Java Card Platform" describes how to develop applications that use RMI on the Java Card platform.

Related Books

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

- *Development Kit User's Guide for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *Application Programming Interface for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *Virtual Machine Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)

- *Runtime Environment Specification for the Java Card™ Platform, Version 2.2.1* (Sun Microsystems, Inc., 2003)
- *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

You can download version 2.2.1 of the Java Card specifications from Sun's web site:

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

Typographic Conventions

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

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`
		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

The Java Developer Connection^{sm} web site enables you to access Java platform technical documentation on the Web:

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

Sun Welcomes Your Comments

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

`docs@java.sun.com`

CHAPTER 1

Using the Object Deletion Mechanism, and Package and Applet Deletion

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

Object Deletion Mechanism

The object deletion mechanism on the Java Card platform reclaims memory which is being used by “unreachable” objects. To be “unreachable”, an object can neither be pointed to by a static field nor by an object field. 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 due to space and time constraints. The amount of available RAM on the card is limited. In addition, since 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. The object deletion mechanism should be used 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();
     }
}
```

Guidelines on Using the Object Deletion Mechanism

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

- When throwing exceptions, avoid creating new exception objects and relying on the object deletion mechanism to perform clean-up. Try to use existing exception objects.

- Similarly try not to create objects in method or block scope. This is acceptable in standard Java, 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 re-sizing is required, as shown in the example in "Requesting the Object Deletion Mechanism" on page 2.

Package and Applet Deletion

Version 2.2.1 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 2.2.1 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.

Guidelines for 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 should have dependencies on the package to be deleted. This includes:

- 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:

- 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 is begun, but before it is completed.

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.

Guidelines for Writing Removable Applets

Deleting an applet means that the applet and all of its child objects are deleted. Applet deletion will not succeed if:

- 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:

- try to 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, try to make sure that these applets can be deleted simultaneously.
- 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, or
  - 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 may implement the `uninstall` method of the `AppletEvent` interface. An applet may 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 may not be deleted after the completion of the `uninstall` method if, for example:

- other applets or packages are still dependent on this applet.

- another applet which needs to be deleted simultaneously cannot be deleted at this time.
- the package which needs to be deleted simultaneously cannot be deleted at this time.
- a tear occurs before the deletion elements are processed.

To ensure that the applets are deleted, implement the `uninstall` method defensively. Write your applet such that:

- 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 2.2.1 of the Java Card platform has the ability to support up to four logical channels. This gives an ISO-7816-4-compliant terminal the ability to open up to four 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 will still work correctly, but they will not be aware of logical channel support. In contrast, applets written for version 2.2.1 can take advantage of this feature.

For example, you could write an applet for version 2.2.1 of the Java Card platform which 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 which is handling session information. Thus, you avoid losing your session-specific security data, which is usually stored in `CLEAR_ON_DESELECT` RAM memory.

Applets and Logical Channels

In version 2.2.1 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 2.2.1 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 2.2.1 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 was 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 with each other. 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.

Multiselectable Applets

If you design your applets to take advantage of multi-session functionality, they will be able to inter-operate with each other from different channels and have the ability to be selected multiple times in different channels. For example, the card might be handling security information on one channel, while data is being 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, then 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 help manage multiselectable applets.

Applet Selection for MultiSelectable Applets

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

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

This method can be called as a result of a `SELECT FILE` or `MANAGE CHANNEL OPEN` APDU command being issued to select an applet. If the applet has not been previously 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.

Applet Deselection for MultiSelectable Applets

```
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. 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, then the `appInstStillSelected` parameter is passed as `true`. Note that if the `MultiSelectable.deselect(boolean)` method is called it basically means that either an instance of this applet or another applet from the same package remains active on another channel; therefore, `CLEAR_ON_DESELECT` transients are not cleared. Only when the last applet instance from the entire package has been deselected, a call to `Applet.deselect()` results instead, resulting in `CLEAR_ON_DESELECT` transients being erased.

Writing Multiselectable Applets

This section describes how to write a multiselectable applet which will perform various tasks based on whether it is selected. The code samples in this section provide examples of extending the applet to implement the `MultiSelectable` interface, and of implementing the `MultiSelectable.select(boolean)` and `deselect(boolean)` methods. The code samples also provide examples of how the `Applet.select()` and `deselect()` methods can be used 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 should have 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 would handle applet selection and deselection behavior when a multiselection situation does not happen.

A 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 that will be initialized when the first applet in the package becomes active
- an array of private applet data that will 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 was selected.

Also, assume that the applet requires you to:

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

and 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 will need to 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, could 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)` would need to determine whether the applet is already active. According to its definition, this method is called when there is another applet within this package that is active. `MultiSelectable.select(boolean)` could be implemented such that if `appInstAlreadySelected` is `false`, then the applet private data could 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 needs to be cleared. The method `MultiSelectable.deselect(boolean)` could be implemented so that it would clear 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 would also need to clear package data. This situation would result in a call to `Applet.deselect()`. This method could 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

Only specific APDU commands can contain encoded logical channel information. These are the commands whose CLA byte contains the bytes 0x0X, 0x8X, 0x9X, and 0xAX.

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), since 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 need to be written to take into consideration a multiple-channel environment, it is important to write the `Applet.process()` method in such a way that it handles channel information correctly. Due to the fact that some APDUs could 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 passed intact to the applet. The applet must deal with this situation.

Note that in the following example, a bit mask, `CHANNEL_FILTER`, is used to filter out logical channel information from the APDU command, so that the applet can correctly interpret it. The task of correctly interpreting `CLA` byte channel information is the responsibility of the applet developer.

```
final static byte CHANNEL_FILTER    = (byte)0xFC;
...

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

    byte[] buffer = apdu.getBuffer();

    // check SELECT APDU command
    if (Applet.selectingApplet()) {
        return;
    }

    // check wallet CLA
    // filter channel information
    if ((byte)(buffer[ISO7816.OFFSET_CLA] &
        CHANNEL_FILTER) != Sample_CLA)
    ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    ...
}
```

Writing ISO 7816-4-compliant Applets

If your applets must be compliant with the ISO 7816-4 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 which channels it is being selected on 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:

- In the `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.

- In the `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.

An ISO 7816-4-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 must 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
        ...
        }
        ...
}
```

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 platform ("Java Card virtual machine" or "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 2.2.1 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 should be made when it is selected.

Working with Non-multiselectable Applets

Applets written for version 2.2.1 of the Java Card platform do not have to implement the `MultiSelectable` interface. In this case, the applet will assume that it is uniquely selected and its owned objects will not be modified via Shareable interface objects while it is selected. The following points describe the limitations that 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 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 will be opened and the specified applet will be 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 provides the flexibility to allow 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 in mind that:

- Origin logical channel values are encoded in the two least significant bits of the CLA byte.
- Logical channel values can be 0, 1, 2, or 3 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, and 3) 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. These functions include:

- managing logical channels
- deselecting applets
- selecting applets

MANAGE CHANNEL OPEN

In response to the `MANAGE CHANNEL OPEN` command, the Dispatcher will follow 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, then the default applet (if there is one) is selected in the new channel.
5. If the origin channel is not 0, then 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 Q:

CLA	**INS**	**P1**	**P2**	**Lc**	**Data**	**Le**	**Data**	**SW1**	**SW2**
0x0Q	0x70	00	00	0	-	1	0x0R	0x90	00

This command will produce the following results:

- the new open channel will be R. The card will select this channel automatically.
- if channel Q is the basic channel (channel 0), the card's default applet will be selected on channel R. No applet will be selected if no default applet is defined.
- if channel Q is other than the basic channel (channels 1, 2 or 3), the selected applet on channel Q will become the current applet selected on channel R.
- applet on channel R can either accept or reject selection.

This command can return an error if:

- 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 available.
- channel Q is not open.

This `MANAGE CHANNEL OPEN` command opens channel R from channel Q:

CLA	INS	P1	P2	Lc	Data	Le	SW1	SW2
0x0Q	0x70	00	0x0R	0	-	0	0x90	00

This command will produce the following results:

- if channel Q is the basic channel (channel 0), then the card's default applet will be selected on channel R. No applet will be selected if no default applet is defined.
- if channel Q is other than the basic channel (channels 1, 2 or 3), then the selected applet on channel Q will become the current applet selected on channel R.
- the applet on channel R can either accept or reject selection.

This command will return an error if:

- the applet does not implement the `javacard.framework.MultiSelectable` interface and you attempt to select it in more than one channel.
- the applet rejects selection or throws an exception.
- channel Q is not open.

MANAGE CHANNEL CLOSE

In response to the `MANAGE CHANNEL CLOSE` command, the Dispatcher will follow 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 will be 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
0x0Q	0x70	0x80	0x0R	0	-	0	0x90	00

This command will close channel R. Channel R must not be the basic channel (it can be channel 1, 2 or 3 only).

This command will return an error if:

- channel R is the basic channel.
- channel Q is not open.

It can return a warning if channel R is not open.

SELECT FILE

In response to the `SELECT FILE` command, the Dispatcher will follow 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 will produce the following results:

- channel R can be any (opened or unopened) channel, including the basic channel.

- the applet identified in the Data section will become the selected applet on channel R.
- if channel R is not open, this command will open channel R.
- if channel R is open, this command will change the selected applet in the channel to the one specified.

This command can return an error if:

- the applet could not 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 R not available.

CHAPTER 3

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 RMI applications in the C-language Java Card RE and the Java Card WDE.

Steps for Developing an RMI Applet for the Java Card Platform

The main steps for developing an RMI applet for the Java Card platform include:

- Define remote interface(s)
- Develop class(es), implementing the remote interface(s)
- Develop the `main` class for the applet

For a simple applet, the main class of the applet could 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 pre-generated and available on the client. When the `remote_ref_with_interfaces` format is used, stubs are not necessary.

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

The command line to run the `rmic` is:

```
rmic -v1.2 -classpath <path> -d <output_dir> <class_name>
```

where:

<*path*> includes the path to the remote class

<*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 – Some versions of the `rmic` do not generate stubs for remote classes which do not list remote interfaces in their `implements` clause. If you encounter this situation, list at least one remote interface in the `implements` clause for each remote class in your project. It is permissible for such a remote class to use a remote interface from a superclass.

The file `javacardframework.jar` is provided in version 2.2.1 of the Development Kit for the Java Card platform ("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 (that is, the Java Card RMI-enabled applet) can be run on both CREF and Java Card platform Workstation Development Environment ("Java Card WDE").

To run the applet on CREF, 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 CREF 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 compiled using `javac` or your favorite IDE. The compiler will require that the remote interfaces for your applet be present in your classpath.

Running the client program requires that:

- OCF1.2 is installed on your workstation, and files `base-core.jar` and `base-opt.jar` must be present in the CLASSPATH.
- the file `opencard.properties` must be present in one of the directories specified in the OpenCard Framework 1.2 Programmers Guide.
- the client framework file `jcrmiclientframework.jar` must be present in the classpath. This file contains all the client framework and necessary classes from the card framework.
- the remote interfaces and stubs for your applet must be present in the classpath.

For a description of the `opencard.properties` file, refer to OCF documentation. A sample `opencard.properties` file is located in the `samples/src/demo` directory Java Card Development Kit, version 2.2.1, and can be used without modification.

For a sample command line to run a client program, refer to the file `rmidemo` or `rmidemo.bat` in this directory.

The `opencard.properties` file supplied in the `samples/src/demo` directory configures the client to use the Java Card WDE or CREF. The RMI client can also use a TLP224-compliant card reader connected to a serial port.

To configure the client to use the card reader, change the `OpenCard.terminals` line in the `opencard.properties` file to the following:

```
OpenCard.terminals =com.sun.javacard.comterminal.ComTerminalFactory|com|COM|<port#>
```

where *`<port#>`* is the name of the serial port, such as `COM1`.

Basic Example

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

The Main Program

As for any Java Card RMI program, the first thing that we need to do is to define the interface that will be used as contract between the server (that is, the Java Card technology-based application) and its clients (that is, 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 interface is a typical Java Card RMI interface:

- 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 in order to encapsulate all the communication problems that may 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 may also define values for constants that may 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 consists in providing an implementation for this interface. This implementation will run 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 and/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 will need to 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:

- 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. The rules for the Java Card platform are:

- 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 may 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, since 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 32.

Note – Even though the semantics of the Java Card platform transient arrays is somewhat similar to transient fields in the Java programming language, different rules apply; its contents are copied in Java Card RMI and passed by value when it is 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 2.2.1 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 which are based on remote objects, for two main reasons:

- the remote objects need to be instantiated and initialized, which can be done in an applet's `install` method.
- the remote objects need to 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.

This 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 needs to 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 needs to 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 code snippet below. 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 amongst several services. In our example, the initialization is quite simple, since there is a single service to initialize:

```
dispatcher = new Dispatcher((short)1) ;
dispatcher.addService(rmi, Dispatcher.PROCESS_COMMAND) ;
```

Finally, the applet needs to 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 will also implement a default behavior for the handling of any request that it does not recognize. In Java Card RMI, there are two kinds of requests that can be handled:

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

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

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

This corresponds to the simplest case, in which the handling of commands is entirely delegated to the dispatcher.

Writing a Client

The client application runs on a terminal supporting a Java Virtual machine environment such as J2SE or J2ME.

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 `com.sun.javacard.javax.smartcard.rmiclient` package.

The example we describe here uses standard Java RMIC compiler generated client side stubs. The client application as well as the Java Card client-side framework rely on the Open Card Framework (OCF) for managing and communicating with the card reader and the card on which the Java Card technology-based applet ("Java Card applet") `PurseApplet` resides. This makes the client application very portable on J2SE platforms.

The OCF can be downloaded from the OCF Web site:

http://www.opencard.org/index-downloads.html

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

```
import opencard.core.service.* ;
import examples.purse.* ;
import com.sun.javacard.javax.smartcard.rmiclient.* ;
import com.sun.javacard.ocfrmiclientimpl.* ;
import javacard.framework.UserException ;

  public class PurseClient extends java.lang.Object {

    /** Creates new PurseClient */
    public PurseClient() {
    }
    public static void main(java.lang.String[] argv) {

      // arg[0] contains the debit amount
      short debitAmount = (short) Integer.parseInt(argv[0]) ;
      try {
        // initialize OCF
        SmartCard.start() ;
        // wait for a smartcard
          CardRequest cr = new CardRequest (CardRequest.NEWCARD,
              null,OCFCardAccessor.class);
          SmartCard myCard = SmartCard.waitForCard ( cr ) ;
          // obtain an RMI Card Accessor CardService for the
          // Java Card platform
          CardAccessor myCS = (CardAccessor)
            myCard.getCardService(OCFCardAccessor.class, true) ;
          // create an RMI connector instance for the Java Card platform
          JavaCardRMIConnect jcRMI = new JavaCardRMIConnect( myCS ) ;
          // select the Java Card applet
          byte[] appAID = new byte[] {0x01,0x02,0x03,0x04,0x05,0x06,0x07, 0x08};
          jcRMI.selectApplet( appAID ) ;
          // 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 {
           SmartCard.shutdown() ;
            }catch (Exception e) {
            e.printStackTrace() ;
          }
        }
    }
}
```

Initializing and Shutting Down OCF

The client application needs to initialize the OCF classes on the terminal. The following code shows this as well as how it is shut down:

```
try {
        // initialize OCF
        SmartCard.start() ;

        // the main client work is performed here
        // ...
}catch (Exception e) {
        e.printStackTrace() ;
} finally {
        try{
        SmartCard.shutdown() ;
        }catch (Exception e) {
           e.printStackTrace() ;
        }
    }
```

Obtaining and Using the OCFCardAccessor Object

To access the Java Card applet using remote methods, the client application needs to obtain an instance of the `CardAccessor` interface. The `OCFCardAccessor` class implements the `CardAccessor` interface and is an OCF card service.

The code below shows how the client interacts with the OCF services. Note that the client specifies that it is requesting a `SmartCard` object on any newly inserted card which supports the `OCFCardAccessor` class:

```
// wait for a smartcard
CardRequest cr = new CardRequest
    (CardRequest.NEWCARD,null,OCFCardAccessor.class) ;
          SmartCard myCard = SmartCard.waitForCard ( cr ) ;

// obtain a Java Card API Accessor for OCF
CardAccessor myCS = (CardAccessor)
          myCard.getCardService(OCFCardAccessor.class, true) ;
```

The `CardAccessor` interface is a platform and framework independent interface which 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 shown below:

```
// 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

In order 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 needs to 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 code below 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 program 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 outlined above 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 JDK1.2 or JDK1.3 RMIC compiler. For example, when in the examples/purse directory, enter:

Solaris and Linux platforms:

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

Microsoft Windows 2000 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 smartcard is first inserted into the card reader, the card sends an ATR (Answer to Reset) to the terminal. The Open Card Framework uses this to help locate the Card Service which may be suitable. In our case, the `OCFCardAccessorFactory`, which has been previously registered with OCF, responds with `OCFCardAccessor` support. FIGURE 1 illustrates this schematically.

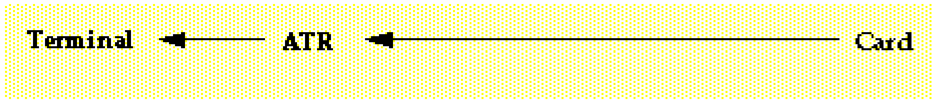

FIGURE 1 Smart card sends an ATR to the terminal

When the `PurseClient` application calls the `selectApplet` method of `JavaCardRMIConnect`, it sends a SELECT APDU Command to the card via the `OCFCardAccessor` 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 which includes the initial reference remote object information. FIGURE 2 illustrates this schematically.

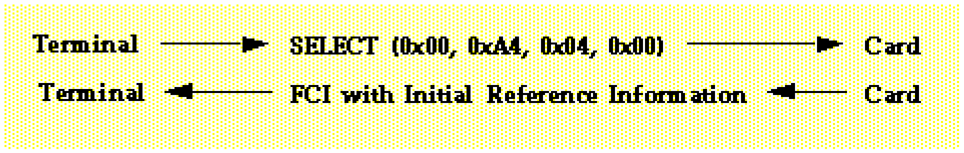
Terminal
SELECT (0x00, 0xA4, 0x04, 0x00)
Card
Terminal
FCI with Initial Reference Information
Card

FIGURE 2 Terminal sends a SELECT command to the smart card. The card returns an FCI to the terminal

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 `OCFCardAccessor` 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. FIGURE 3 illustrates this schematically.

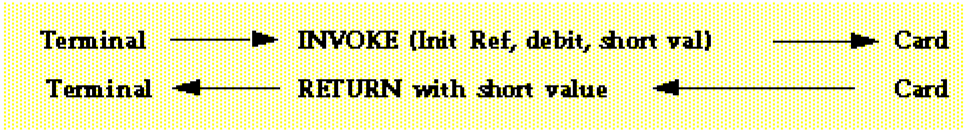

FIGURE 3 Terminal sends an INVOKE command to the smart card. The card returns a value to the terminal.

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 will include such security mechanisms, since they are central to Java Card technology.

In the following section, you will see 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. It should be replaced with `examples.securepurse`. The interface does not include any implementation, which means that, in particular, it does not include any support for 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 ;
  }
}
```

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

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. Since a generic remote object should not be dependent on a particular kind of security service, it should 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 required security behavior for the applet is assumed:

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

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

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

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

Implementing 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 needs to 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 needs to 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, we need to initialize the dispatcher. Here, it needs to not only dispatch simple Java Card RMI requests, but also to dispatch security-related requests (such as EXTERNAL AUTHENTICATE). In fact, the security service will handle these requests directly. We first need to allocate a dispatcher and to inform it that it will delegate commands to two different services:

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

Then, the services are "registered" with the dispatcher: the security service as a service that performs pre-processing operations on incoming commands, and the Java Card RMI service 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 (that is, the 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 does additionally need to interact with the user for identification. Hence, a subclass of `OCFCardAccessor` needs to be developed which provides these additional interactions and the transport filtering required.

Here is the new `SecurePurseClient` application:

```
package examples.securepurseclient ;

import com.sun.javacard.javax.smartcard.rmiclient.* ;
import com.sun.javacard.clientsamples.securepurseclient.SecureOCFCardAccessor ;
import opencard.core.service.* ;
import examples.securepurse.* ;
import javacard.framework.UserException ;

public class SecurePurseClient extends java.lang.Object {

// need to authenticate user with the secure purse applet
// prompt user for account and password information
// to initialize user key and card info from database
private final static short PRINCIPAL_CARDHOLDER_ID=0x4321

    /** Creates new SecurePurseClient */
    public SecurePurseClient() {
    }
    public static void main(java.lang.String[] argv) {
        // arg[0] contains the debit amount
        short debitAmount = (short) Integer.parseInt(argv[0]) ;
        try {
          // initialize OCF
          SmartCard.start() ;
          // wait for a smartcard
          CardRequest cr = new CardRequest ( CardRequest.NEWCARD,
                null, SecureOCFCardAccessor.class ) ;
          SmartCard myCard = SmartCard.waitForCard ( cr );
          // obtain a custom RMI CardAccessor class for
          // the Java Card platform: SecureOCFCardAccessor
          SecureOCFCardAccessor myCS =
                  (SecureOCFCardAccessor) myCard.getCardService(
                                SecureOCFCardAccessor.class, true ) ;

            //create RMI connection for the Java Card platform
            JavaCardRMIConnect jcRMI = new JavaCardRMIConnect(myCS) ;
            // select the Java Card applet
            byte[] appAID = new byte[] {0x01,0x02,0x03,0x04,0x05,0x07} ;
            jcRMI.selectApplet( appAID );

            if (! myCS.authenticateUser( PRINCIPAL_CARDHOLDER_ID )) {
              // handle error
             }
            // obtain the initial reference to the Purse interface
            Purse myPurse = (Purse) jcRMI.getInitialReference() ;
            // debit the requested amount
            try {
                // the debit invocation command will be signed
                // by SecureOCFCardAccessor
                short balance = myPurse.debit ( debitAmount ) ;
            }catch ( UserException jce ) {
                    short reasonCode = jce.getReason() ;
                    // process on card exception reason information
            }
            // display the balance to user
        }catch (Exception e) {
          e.printStackTrace();
        } finally {
            try{
            SmartCard.shutdown();
            }catch (Exception e) {
                 e.printStackTrace();
            }
        }
    }
}
```

Note how the `SecureOCFCardAccessor` instance is now obtained instead of `OCFCardAccessor`:

```
CardRequest cr = new CardRequest ( CardRequest.NEWCARD,
                      null, SecurePurseClientCardService.class ) ;
SmartCard myCard = SmartCard.waitForCard ( cr );
    // obtain a customized Card Accessor class: SecureOCFCardAccessor
    SecureOCFCardAccessor myCS =
    (SecureOCFCardAccessor) myCard.getCardService(
        SecureOCFCardAccessor.class, true ) ;
```

An extra step to authenticate with the SecurePurseApplet after selectApplet has been added. This invokes a new method in SecureOCFCardAccessor to interact with the card using the users credentials:

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

The rest of SecurePurseClient is the same as PurseClient.

Writing a Custom SecureOCFCardAccessor Class

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

```
package examples.securepurseclient;

import opencard.core.terminal.* ;
public class SecureOCFCardAccessor extends
              com.sun.javacard.ocfrmiclientimpl.OCFCardAccessor {
      /** Creates new SecureOCFCardAccessor */
    public SecureOCFCardAccessor() {
    }
  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
    }
}
```

As shown above, the `SecureOCFCardAccessor` 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 custom Java Card RMI Card Accessor class also re-implements the `exchangeAPDU` method declared in a superclass `OCFCardAccessor` to sign each message before it is sent out by `super.exchangeAPDU`.

Index