Java Card 3 Platform
Programming Notes

Classic Edition,Version 3.0.5
E59598-02
April 2020

ORACLE

Java Card 3 Platform Programming Notes, Classic Edition,Version 3.0.5

E59598-02

Contents

Glossary

Preface

This book contains tips and guidelines for developers of Java Card applets and for developers of vendor-specific frameworks. This book covers several topics that are substantially different from programming models found in earlier versions of the Java Card platform and is not meant to comprehensively introduce or cover general programming topics.

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).

Audience

This book is for applet developers using the *Application Programming Interface for the Java Card Platform, Version 3.0.5, Classic Edition* to implement applet management, multiselectable applets, logical channels, 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 3.0.5 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.

You must also be familiar with the development tools released with version 3.0.5 of the Java Card platform. For information on these tools, see the *Java Card 3 Platform Development Kit User's Guide, Classic Edition Version 3.0.5*.

How This Document Is Organized

The chapters and appendices in this guide are described in the following list:

- Object_ Package and Applet Deletion describes how to perform object deletion, applet deletion, and package deletion on the Java Card 3 platform.
- Working with Logical Channels describes how to create and use applets that can be selected for use on multiple channels on the Java Card 3 platform.
- Developing RMI Applications for the Java Card Platform describes how to develop applications that use the optional RMI APIs on the Java Card 3 platform.

- Using Extended APDU describes how to handle extended APDU functionality on the Java Card 3 platform.
- Glossary defines terms used in the Java Card 3 Platform.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

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

- *Java Card 3 Platform Development Kit User's Guide Classic Edition, Version 3.0.5*
- *Java Card 3 Platform Runtime Specification, Classic Edition Version 3.0.5*
- *Java Card 3 Platform Virtual Machine Specification, Classic Edition Version 3.0.5*
- *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/IEC 7816 Specification* Parts 1-6.

Conventions

The following text conventions are used in this document:

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

1

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 3 Platform.
This chapter includes the following topics:

Object Deletion Mechanism

The object deletion mechanism on the Java Card 3 Platform reclaims memory that is being used by "unreachable" objects. Objects become unreachable for a number of reasons such as static or instance fields having missing pointers, missing variable references (not only fields), or when the object is orphaned in an island of isolation. An applet object is reachable until it is successfully deleted.

The object deletion mechanism 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 the 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.

Requesting the Object Deletion Mechanism

Although any applet on the card can request it, only the Java Card Runtime Environment (Java Card RE) can start the object deletion mechanism. The applet requests the object deletion mechanism with a call to the `JCSystem.requestObjectDeletion()` method.

In the following code example, the 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

The following guidelines describe 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.
- 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.

Package and Applet Deletion

In the Java Card 3 platform, the installer deletes packages and applets from the card's memory. Once the installer is selected, it can receive requests from the terminal, in the form of an APDU, to delete packages and applets. Requests to delete an applet or package cannot be sent from an applet on the card.

The following sections describe programming guidelines that will help you create packages and applets that are more easily removed:

- Developing Removable Packages
- Writing Removable Applets

Developing Removable Packages

When a package is deleted, all of its code is removed from the card's memory. A package is eligible for deletion only if there are no dependencies on it, including:

- 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 easily removed from the card, avoid writing and downloading other packages that might be dependent on it. If other packages on the card depend on it, you must remove all 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 operation, follow these guidelines to ensure that the applet can be easily removed:

- 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 the following 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).

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;
    }
    ...
}
```

2
Working with Logical Channels

The Java Card 3 Platform can support up to twenty logical channels per active interface. Logical channels allow the concurrent execution of multiple applications on the card, allowing a terminal to handle different tasks at the same time.
This chapter includes the following topics:

Dual Interface Cards

On dual interface cards, each interface can handle up to twenty independent logical channels. Channel management commands only affect the logical channels in the interface where the commands are issued.

For more information on logical channels, their implementation and logical channel terminology, see the *Java Card 3 Platform Runtime Environment Specification, Classic Edition Version 3.0.5*.

Applets and Logical Channels

If you design your applets to take advantage of multi-session functionality, they can interoperate 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.

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.

Non-MultiSelectable Applets

An error is returned to the terminal when an applet that is not designed to be aware of multiple channels is either selected more than once on different channels or is selected concurrently with other applets in the same package.

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 interoperate. 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.

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.

> **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. These methods are described in the following topics:

- Selection for MultiSelectable Applets
- Deselection for 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 a `MANAGE CHANNEL OPEN` APDU command used 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.deselect()` 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, indicating 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 a `SELECT FILE` APDU command. If the applet remains active on a different channel, the `appInstStillSelected` parameter is passed as `true`.

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

Related Topics

- MultiSelectable Applet Example
- Handling Channel Information on APDU Commands
- Writing ISO 7816-4:2005 Compliant Applets
- Non-MultiSelectable Applets and Shareable Objects
- ISO 7816-4:2005 Specific APDU Commands for Logical Channel Management

MultiSelectable Applet Example

In this example, 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, 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

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 so that if `appInstAlreadySelected` is `false`, 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/IEC 7816-4:2013 specifications to encode logical channel information in the CLA byte. The CLA byte encoding is divided into two spaces:

- Interindustry —Used by all ISO/IEC 7816-4:2013- defined commands
- Proprietary — Used by Java Card technology to encode application- specific commands

The CLA byte encoding is divided into two classes:

- Type 4 commands — Encode legacy ISO/IEC 7816-4 logical channel information
- Type 16 commands — Defined by the ISO/IEC 7816-4:2013 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 the `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

- Proprietary Java Card Technology Space
- Logical Channels
- APDU Command Type Identification

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` Reserved for Future Use

`0x3X` Reserved for Future Use

`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 `0x06X`)

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

Logical Channels

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`).

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`. For example, the command `0x21` forwards the command to the card's basic channel (`0`), because the CLA byte with the nibble `0x2X` does not contain logical channel information.

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.

APDU Command Type Identification

To identify proprietary and interindustry commands, use the `isISOInterindustryCLA` method. 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/IEC 7816-4:2013 Compliant Applets

If your applets must be compliant with the ISO/IEC 7816-4:2013 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)` methods, it returns the APDU command logical channel 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)` methods, it returns the channel where the applet to be selected or deselected is assigned to run.

ISO/IEC 7816-4:2013 Compliant Applet Example

This example demonstrates how to copy the security state from the applet selected in the origin channel into the new channel.

In this example, 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 the `JCSystem.getAssignedChannel()` methods to identify if the applet selection case corresponds to an ISO/IEC 7816-4 case where the security state needs to be copied.

> **Note:**
>
> 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
        ...
    }
    ...
}
```

Refer to the API documentation in the in *JC_CLASSIC_HOME*`\docs` for more information about the APIs.

Non-MultiSelectable Applets and Shareable Objects

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 that does not implement `MultiSelectable` assumes that it is uniquely selected and its owned objects will not be modified via Shareable interface objects while it is selected. Only when the non-multiselectable applet is in a deselected state can other applets modify its internal data structures.

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/IEC 7816-4:2013 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	SW2
0xQ	0x70	00	0xR	0	-	0	0x90	00	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	SW2
0xQ	0x70	0x80	0xR	0	-	0	0x90	00	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.

3

Developing RMI Applications for the Java Card Platform

This chapter describes how to write remote method invocation (RMI) applications for the Java Card 3 Platform. Because the Java Card specifications state that Java Card RMI is optional, verify that your targeted card supports Java Card RMI before using these APIs.
This chapter includes the following topics:

Steps to Develop an RMI Applet for the Java Card 3 Platform

There are three main steps to develop an RMI applet:

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.

This section includes the following topics:

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, the Java 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*

In the command:

- *path* includes the path to the remote class of your sample applet and to the file `tools.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 3 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 `tools.jar`, provided in the Java Card Development Kit contains compiled implementations of packages `javacard.framework`, `javacard.security`, `javacardx.biometry`, `javacardx.external` and `javacardx.framework.tlv`. 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 the C-language Java Card RE, for which the following standard procedures apply:

- The applet must be installed first by using the installer applet.
- After the applet is installed, the EEPROM state can be saved and used to run the Java Card RE against the Java Card RMI client.

RMI Program Example

The RMI program example is the Java Card platform equivalent of "Hello World." It is a program that manages a counter remotely, and is able to decrement, increment, and return the value of the counter.

This section includes the following topics:

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 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.

Related Topics

- Implement a Remote Interface
- Define the Constructor for the Remote Object
- Provide an Implementation for Each Remote Method

Implement a Remote Interface

This code sample provides an implementation for the remote interface. The implementation runs on a Java Card 3 platform, so it can use only features that are supported by a Java Card 3 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 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 needs to be exported to make 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, the following code 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 3 platform does not support object serialization. The following are the rules for the Java Card 3 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 3 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 3 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 3 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.

  > **Note:**
  >
  > Even though the semantics of the Java Card 3 platform transient arrays are somewhat similar to transient fields in the Java programming language, different rules apply. The Java Card 3 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.

Sample Applet

In the Java Card 3 platform, 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.

The 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);
    }
}
```

Related Topics

- Preparing and Registering the Remote Object
- Processing the Incoming Commands

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

Processing 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 can be handled:

- **Selection request** — The service responds by sending its initial remote reference
- **Method invocation request** — 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);
```

Client Example

Client applications run on a terminal supporting a Java Virtual Machine environment such as Java Platform, Standard Edition or Java Platform, Micro Edition (Java ME).

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 client example below 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 Java SE platforms. See the *Java Card 3 Platform Development Kit User Guide, Classic Edition Version 3.0.5* for information on the APDU I/O library.

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

```
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();
             }
```

```
        }
    }
}
```

Related Topics

- Initializing and Shutting Down the Card Connection
- Creating and Using a CardAccessor Object
- Selecting the Java Card Applet and Obtaining the Initial Reference
- Using Remote Objects in Remote Method Invocations
- Generate the Stubs

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`. For example. when the RMIPurse sample demo client application runs, the *JC_CLASSIC_HOME*`/samples/classic_applets/RMIPurse/client` directory containing the properties file is included in the `CLASSPATH`. The directory in which you installed the developer's kit is indicated as *JC_CLASSIC_HOME*.
>
> On Microsoft Windows platforms, use backslashes in directory paths, instead of forward slashes.

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 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 dialog for the Java Card platform as shown in the following code:

```
// 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 `PurseApplet` on the card, it must first be selected by using the AID as shown in the following code:

```
// 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 as shown in the following code:

```
// 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:**
>
> A `UserException` exception thrown by the remote method is caught by the client code in 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. 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 JDK compiler. The directory where you installed the developer's kit is indicated by `JC_CLASSIC_HOME`. For example, from the `examples/purse` directory, enter the following command:

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

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

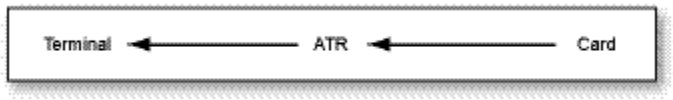
For `PurseClient` to run correctly on the terminal, the following files must be present in the `examples/purse` directory and accessible either from its classpath or from class loaders:

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

Card Terminal Interaction

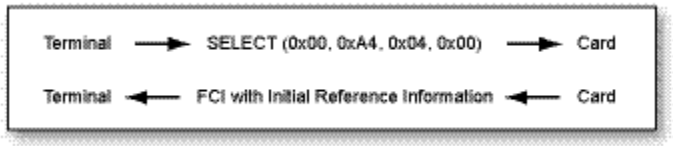
When a Java Card technology-enabled smart card is powered up, the card sends an ATR (Answer to Reset) to the terminal. The Card Accessor returns the value of the ATR to the client program (shown in Figure 3-1).

Figure 3-1 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 (shown in Figure 3-2).

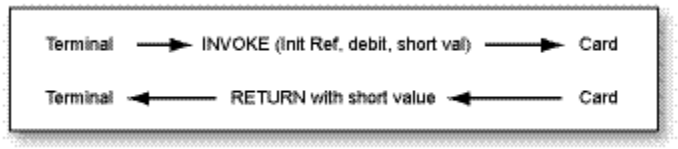
Figure 3-2 Terminal Sends a SELECT Command to the Smart Card, Which Returns FCI



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 `RETURN` APDU (shown in Figure 3-3).

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



Add Security Support

The previous Sample Applet example is extremely simple and is not realistic. In particular, it does not include any form 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 used in the Sample Applet example. In addition, it might include reason codes for exceptions to report security violations to the terminal. This example replaces the `Purse` interface used in theSample Applet example with the following `examples.securepurse` code. The`Purse` interface in the `examples.securepurse` does not include an 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_CARDHO
LDER)))
           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_PRO
VIDER)))
           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_PRO
VIDER)))
            UserException.throwIt(REQUEST_FAILED);
        return balance;
    }
}
```

Related Topics

- Initialize a Security Service
- Use the Service to Check the Current Security Status
- Security Service Example
- More Secure Applet
- Client Changes to Support Security
- CustomCardAccessor Class for Authentication and Signing

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, the following are required security behaviors for the applet:

- 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.

Security Service Example

The following example demonstrates 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
     }
}
```

More Secure 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.

Client Changes to Support Security

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.

The following code 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();
            // create an RMI connector instance for the Java Card platform
            JCRMIConnect jcRMI = new JCRMIConnect(cca);
            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(cca!=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`.

CustomCardAccessor Class for Authentication and Signing

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`.

4

Using Extended APDU

This chapter describes the Extended APDU and how it can be used to allow large amounts of data to be sent to the card, processed appropriately, and sent back to the terminal.
The Extended APDU feature is especially beneficial to applications that deal with large amounts of information, such as signature verification, biometrics verification and image storage and retrieval. These are more easily implemented if the underlying transport protocol is T=1. Applets developed for T=0 cards need special logic and care to work correctly.

This chapter includes the following topics:

Extended APDU Nominal Cases

The ISO/IEC 7816-4:2013 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. 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

Any APDU classified as extended must follow the format defined by ISO/IEC 7816-4:2013 for extended length APDU and summarized in Table 4-1.

Table 4-1 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 3 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 3 platform at this time.

This section includes the following topics:

- javacardx.framework.ExtendedLength Interface
- APDU Parsing with the javacard.framework.APDU Class

javacardx.framework.ExtendedLength Interface

By implementing the `javacardx.apdu.ExtendedLength` interface, applets indicate that they are capable of processing, receiving, and replying to extended APDU commands. The Java Card RE does not deliver extended APDU commands to applets that do not implement this interface (it would throw an `ISOException` with reason code `ISO7816.SW_WRONG_LENGTH`). In addition, the Java Card RE does not allow applets to send reply data lengths greater than 256, if the interface is not implemented by the applet.

The APDU buffer in Java Card applications reflects the structure of the extended APDU as defined in the ISO/IEC 7816-3 specification. In T=1, this representation is straightforward and precise; however, in T=0, adaptations are needed 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 interpreted as 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/IEC 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.

APDU Parsing with the 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`.

To get the value of LC and the data offset inside the APDU buffer use these two APIs in `javacard.framework.APDU`:

- `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.

Creating an Applet That Can Send and Receive Extended Length APDUs

To create an applet that can send and receive extended length APDUs:

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. Use the API 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
}
```

Glossary

active applet instance

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

AID (application identifier)

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

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

APDU

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

applet

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

applet developer

a person creating an applet using Java Card technology.

applet execution context

context of a package that contains currently active applet.

applet firewall

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

applet package

see library package.

assigned logical channel

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

atomic operation

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

atomicity

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

ATR

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

basic logical channel

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

binary compatibility

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

CAD

an acronym for Card Acceptance Device. A CAD can integrate a card reader and is the device in which the card is inserted. Sometimes the CAD is called the card reader.

CAP file

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

CAP file component

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

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

card session

a card session begins with the insertion of the card into the CAD. The card is then able to exchange streams of APDUs with the CAD. The card session ends when the card is removed from the CAD.

constant pool

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

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

context

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

Converter

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

currently selected applet

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

custom CAP file component

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

default applet

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

EEPROM

an acronym for Electrically Erasable, Programmable Read Only Memory.

entry point objects

see Java Card RE entry point objects.

Export file

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

finalization

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

Finalization is not supported by the Java Card virtual machine. There is no `finalize()` method to be called automatically by the Java Card virtual machine.

firewall

see applet firewall.

flash memory

a type of persistent mutable memory. It is more efficient in space and power than EEPROM. Flash memory can be read bit by bit but can be updated only as a block. Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole.

framework

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

garbage collection

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

heap

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

installer

the on-card mechanism to download and install `CAP` files. The installer receives executable binary from the off-card installation program, writes the binary into the

smart card memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card Runtime Environment.

installation program

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

Java Card Platform Remote Method Invocation

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

Java Card Runtime Environment (Java Card RE)

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

Java Card Virtual Machine (Java Card VM)

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

Java Card RE entry point objects

objects owned by the Java Card RE context that contain entry point methods. These methods can be invoked from any context and allow non-privileged users (applets) to request privileged Java Card RE system services. Java Card RE entry point objects can be either temporary or permanent:

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

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

library package

a Java programming language package that does not contain any non-abstract classes that extend the class `javacard.framework.Applet`. An applet package contains one or more non-abstract classes that extend the `javacard.framework.Applet` class.

logical channel

as seen at the card edge, works as a logical link to an application on the card. A logical channel establishes a communications session between a card applet and the terminal. Commands issued on a specific logical channel are forwarded to the active

applet on that logical channel. For more information, see the *ISO/IEC 7816 Specification, Part 4*. (http://www.iso.org).

mask production (masking)

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

multiselectable applets

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

multiselected applet

an applet instance that is selected and, therefore, active on more than one logical channel simultaneously.

namespace

a set of names in which all names are unique.

native method

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

nibble

four bits.

normalization (classic applet)

the process of transforming and repackaging a Java application packaged for the Java Card Platform, Version 2.2.2, for deployment on the Java Card 3 Platform.

normalization (URI)

the process of removing unnecessary "." and ".." segments from the path component of a hierarchical URI.

Normalizer

a software tool that allows Java applications programmed for the Java Card Platform, Version 2.2.2, to be deployed on both the Java Card 3 Platform, Connected Edition and on the Java Card 3 Platform, Classic Edition. It also allows Java applications packaged for Version 2.2.2 to be transformed through the normalization process and then repackaged for deployment on both the Connected and Classic Editions.

object owner

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

origin logical channel

the logical channel on which an APDU command is issued.

PCD

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

persistent object

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

PIX

see AID (application identifier).

RAM (random access memory)

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

reference implementation

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

remote interface

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

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

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

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

remote methods

the methods of a remote interface.

remote object

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

RID

see AID (application identifier).

RMI

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

ROM

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

runtime environment

see Java Card Runtime Environment (Java Card RE).

shareable interface

an interface that defines a set of shared methods. These interface methods can be invoked from an applet in one context when the object implementing them is owned by an applet in another context.

shareable interface object (SIO)

an object that implements the shareable interface.

smart card

a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smart cards carry both processing power and information. They do not require access to remote databases at the time of a transaction.

terminal

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

thread

the basic unit of program execution. A process can have several threads running concurrently each performing a different job, such as waiting for events or performing a time consuming job that the program doesn't need to complete before going on. When a thread has finished its job, it is suspended or destroyed.

The Java Card virtual machine can support only a single thread of execution. Java Card technology programs cannot use class `Thread` or any of the thread-related keywords in the Java programming language.

transaction

an atomic operation in which the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

transient object

the state of transient objects do not persist from one CAD session to the next, and are reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions.

verification

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