Java Card 2.0 Programmers Guide

1. Introduction

This guides describes how to use the Java Card 2.0 API to write applets for smart cards.

2. Applets

According to the ISO 7816-4 standard, an ICC (Integrated Circuit Card with contacts), also known as a "smart card," contains one or more "applications." Java Card uses the term "applet" in place of the ISO term "application."

Applets are the basic unit of selection, context, and functionality. When a smart card is inserted in a CAD (Card Acceptance Device), the CAD will select an applet on the card and send it commands to perform. Applets are identified and selected by an AID (Application IDentifier) as defined in ISO 7816-5. The selection and other commands are formatted and transmitted as APDUs (Application Protocol Data Units) as defined in ISO 7816-4. Applets reply to each APDU command with optional data and a SW (Status Word) indicating the result of the operation.

2.1 Multiple Applets

A CAD may interact with a single applet. Or it may interact with several applets by selecting each in turn and sending them individual APDU commands to perform. Each applet is an independent entity with its own state and functionality. Under normal circumstances, the existence and operation of one applet has no effect upon other applets on the card. However, Java Card provides facilities to support more sophisticated scenarios whereby multiple applets can discover each other, communicate, and share data in a limited fashion, while still maintaining reasonable protection from each other.

2.2 Packages

Java Card supports packages as in standard Java. Packages, like applets, are named with AIDs. A minimal applet is a package with a single class derived from the javacard.framework.Applet class. On the other hand, an applet may consist of code in multiple packages downloaded as a unit and linked to other packages already on the card. Packages not containing an applet class can also be loaded onto the card.

2.3 Object Lifetime

In a Java Card system, applets are written in the Java language, and they make use of Java "objects" to represent, store, and manipulate data. Each applet has its own object space in which to create objects. Unless an object is explicitly shared, it is only accessible to the applet which created it.

Java Card objects persist across card uses (insertion of the card into a reader). In a PC or Workstation, the Java virtual machine runs as an operating system process. When the OS process dies all Java applets and their objects are destroyed. Because of the persistent memory technology (like EEPROM) used throughout the smart card industry, however, Java Card assumes that the virtual machine process lives for the life of the card. So Java Card objects have the same lifetime as the virtual machine.

If all references to an object are forgotten, then the object itself is lost. However, garbage collection is not performed, and the space occupied by that object is not reclaimed[1]. Thus, the nature of Java Card applet design is somewhat different from that of application design for other Java environments.

On a PC, for example, where there is lots of RAM, a fast processor, and a native file system fronting a large disk, Java programs typically create large numbers of objects during processing. Many objects are also routinely “forgotten,” and these are periodically garbage-collected so that their space can be reused.

Java Card platforms are much more limited in RAM, processor speed, and storage space. Java Card applets must be designed with this in mind. The Java Card object system is much like an Object DataBase, compared to the transient nature of the object system of a Java VM on a PC.

Java Card applets must be conservative and careful in their creation of objects in order to avoid wasting precious memory resources.

2.4 Transient Objects

Most object data maintained by an applet needs to be persistent across “sessions” with the CAD. Due to the nature of smart card hardware, there is a performance penalty when updating objects in some memory technology.

But applets sometimes have objects that contain temporary or transient data which need not be persistent across sessions. Thus, applet performance can be increased if transient data is collected into one of more transient objects.

Java Card supports designating objects as “transient.” These objects are like the normal Java Card objects in all ways, except for the following:

- The contents of transient objects are not preserved across CAD sessions.
- Transient objects are potentially much faster to update than persistent objects.
- Due to resource limitations, the number of transient objects an applet can have is quite small.
- Transient objects do not have any limitations with respect to the number of times they can be written (unlike persistent memory technology such as EEPROM).

Thus, transient objects are ideal for the small amounts of applet temporary data which is modified frequently during the course of a session but need not be preserved across sessions.

---

[1] Garbage collection may be implemented in future, but it is not practical on today’s limited smart card platforms.

A transient object's space and object reference is still persistent because, once created, they take up space as long as the virtual machine. But while the object itself is persistent, the contents of the object's fields are not.

Like persistent objects, Java Card applets must be conservative and careful in their creation of transient objects in order to avoid wasting precious memory resources.

2.5 Atomicity

The Java Card platform guarantees that any update to a single object or class field will be "atomic." That is, if the smart card loses power during the update, the contents of the field will be restored to its previous value. Some Java Card methods also guarantee atomicity for "block updates" of multiple data elements. For example, the atomicity of the `copyArray` method guarantees that either all bytes are correctly copied or else the destination array is restored to its previous byte values.

In some cases, an applet will need to atomically update several different fields in several different objects. That is, either all updates take place correctly or else all fields are restored to their previous values.

Java Card supports a "transactional model" whereby an applet can designate the beginning of an atomic set of updates with a call to the method `beginTransaction`. Each object update after this point is "conditionally updated." This means that the field appears to be updated – reading the field back yields its latest conditional value. But the update is not yet "committed."

When the applet calls `commitTransaction` then all conditional updates are committed to persistent storage.

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

Note that only updates to persistent objects participate in the transaction. Updates to transient objects are never undone, regardless of whether or not they were "inside a transaction."

Since platform resources are limited, the number of bytes of conditionally updated data that can be accumulated during a transaction is limited. Java Card provides methods to determine how much "commit capacity" is available on the current platform. An error is throw if the commit capacity is exceeded.

2.6 Security and Object Sharing

In order to create a secure and trusted environment, applets are isolated from each other. By default, any object (persistent or transient) created by one applet cannot be accessed nor even seen by any other applet on the card.

However, in order to support cooperating applets, Java Card allows an applet to explicitly "share" any of its objects with one or more other applets. These additional applets can now access (read and write) the fields and call methods of the shared object. The granularity of sharing is on an object and applet basis. Individual objects are explicitly shared with one or more individual applets. This gives the applet developer very fine-grained control over object sharing.

For convenience in handling "global data," it is possible for an object to be shared to "all" applets on the card.

2.7 Exception Handling

Java Card supports exception handling as defined in the Java language. Exceptions are thrown both by the VM when internal runtime problems are detected. In addition, exceptions can be thrown programmatically by the code in applets and shared packages. Exceptions are caught in the normal Java way.

"Checked" exceptions (see *The Java™ Language Specification*, section 11.2) are subclasses of `CardException` and must either be caught in the throwing method or declared in a `throws` clause of the method header. These exceptions are typically an important part of the interface to a method and must be eventually caught by the applet code in order to ensure correct usage of the API.

"Unchecked" exceptions are subclasses of `CardRuntimeException` and need not be `caught` nor declared in a `throws` clause. These exceptions are typically indicative of unexpected runtime problems or programming errors and are caught by the outmost levels of the Java Card system. (However, an applet may catch unchecked exceptions if it chooses to do so.)

One major difference between exception handling in Java Card and in other Java platforms results from the fact that all Java Card objects are persistent.

On a PC Java platform, exception objects are usually dynamically created (with a `new`) when they occur. After they are caught and handled, the exception objects are then forgotten and garbage-collected.

This works perfectly well on a platform with lots of RAM and garbage collection. But, as we have seen, Java Card platforms do not have large amounts of RAM, garbage collection is not done, and objects persist.

When a Java Card exception object is created, it persists and continues to occupy precious memory space, even if its reference is later forgotten. Thus, the Java Card paradigm is to "pre-create" all exception objects at some initialization time and save their references permanently in some well-known location. When the exception event occurs, rather than create a `new` exception object, the code should retrieve the reference for the desired exception object from the well-known location, fill in parameter fields in the object with any exception event information, and throw the object.

When the exception is eventually caught and handled, the reference can be forgotten by the handling code. But, of course, the exception object is never truly forgotten because it is still referenced in the well-known location. Thus, the next occurrence of that exception will reuse this same object.

To reiterate, in Java Card one instance of each class of exception should be pre-created and saved. This instance should be reused whenever it needs to be thrown. This avoids wasting memory when generating exceptions.

The Java Card system pre-creates an instance of each kind of specific exception defined in the Java Card API. Most of these are unchecked exceptions. Whenever they are needed, the references to these objects can be obtained by applet code with the `javacard.system.getxxxException` methods.

Applets may define their own exceptions by subclassing `UserException`. These are always checked exceptions. These exceptions can be thrown and caught as desired by the applet. However, during initialization the applet should create a single instance of each such exception, save the reference in some persistent object field, and reuse that instance whenever it is necessary to throw that exception.

2.8 Temporary Objects

The Java Card paradigm for exceptions also applies to any other kind of "temporary" object that an applet may wish to use. As noted, if applets were to dynamically create temporary objects as needed and then forget them, the platform memory would quickly become exhausted, since the space for these objects is not reclaimed.

Instead, applets should pre-create (at initialization time) all the instances of all temporary objects that the applet may need during its lifetime. These references should be persistently stored in a known location, and then (re)used as necessary.

2.9 Applet Lifetime and Runtime Environment

Since there is no garbage collection, and since applets are simply a collection of Java Card objects, applets themselves persist for the life of the card. In other words, once installed an applet lives on the card forever[2].

Each applet is a subclass of the `Applet` class. As defined by this template, an applet must implement for the methods `install`, `select`, and `process`

The process of developing, distributing, and installing applets is beyond the scope of this document. For the purposes of this document, we begin an applet's lifetime at a point where it has been correctly loaded into platform memory, linked, and otherwise prepared for execution. The last phase of this installation process is that the Java Card system calls the `install` static method of the `Applet` class.

2.9.1 install()

When `install` is called, no applet objects exist. The main task of the `install` method within the applet is to create and initialize the objects that the applet will need during its lifetime and otherwise prepare itself to be selected and accessed by a CAD.

[2] If desirable, the Java Card API could implement a method to allow the system to "forget" an applet. This would have the effect of "deleting" the applet from the card. But the space for that applet would not be reclaimed. Such a method is not currently included in the Java Card API. Furthermore, true applet deletion (including garbage collection) is a possible enhancement for a future version of Java Card.

Depending on the platform constraints and the applet design, there is quite a lot of flexibility in what can be done during the `install` method. Typically, an applet will create various objects (persistent and/or transient), initialize them with predefined values, link them together in some fashion, set some internal state variables, and call the `register` method to inform the Java Card system that the applet is available for selection.

The actual install APDU command is supplied as an `install` parameter, so that the applet may examine parameter data describing how to configure itself. Should the applet encounter a problem with installation, it may return from `install` with an error SW (anything other than 0x9000) and the system will abort the applet installation.

Simple applets may be fully ready to function in their normal role after a successful return from `install`.

More complex applets may need further configuration, initialization, or personalization information before they are ready to function normally. In this case, the applet should set internal state variables indicating that the next few APDUs it receives must deliver the additional data necessary to complete the installation (see the `select` and `process` methods below).

2.9.2 Use of new()

The Java Card system allows a card issuer to specify its own policies concerning the operation of various facets of the platform. One of these policies is whether or not applets are allowed to perform a `new` after installation.

Some card issuers may choose to enforce a policy whereby applets may only create new objects ("`new`") during the installation process (between the first call to the applet's `install` method and the applet's call to the system's `register` method).

Other card issuers may adopt a more lenient policy whereby applets may create new objects ("`new`") at any time during their lifetime (subject, of course, to the availability of platform memory). Applet developers should check with card issuers to determine the policies for each issuer platform.

In any case, all `news` must always be enclosed within a transaction. This requirement is enforced by the Java Card system and is necessary in order to protect the integrity of internal Java Card system data structures, as well as help the applet ensure that new objects are correctly linked into the applet's environment and are not "forgotten."

2.9.3 select()

Applets remain in a "suspended" state until they are explicitly selected. Selection occurs when the Java Card system receives a SELECT APDU where the name data matches the AID of the applet. Selection causes this applet to become "active," and the platform context is adjusted so that only objects belonging to this applet (or appropriately shared to this applet) can be accessed. Finally, the system informs the applet of selection by invoking its `select` method. In this case, the `selectedFlag` parameter is true, indicated that this applet has just become active.

The actual install APDU command is supplied as a `select` parameter, so that the applet may examine parameter data (if any). The applet may respond to the select APDU with data (see the `process` method for details) and returns from `select` with a SW. The SW and optional response data are sent to the CAD. Any SW returned from `select` other `0x90nn` indicates refusal of the applet to be selected, and this applet will no longer be "active."

After successful selection, all subsequent non-select APDUs are delivered to this applet via the `process` method. If a select APDU contains the name of another applet (or even this same applet!) this applet become "inactive." The newly identified applet becomes active and its `select` method is called.

If a select APDU contains a name which is not recognized by the Java Card system as the AID of an applet, then the `select` method of the active applet is called. In this case, the `selectedFlag` parameter is false, indicated that this applet is already active and has received a select APDU which does not correspond to a known applet on the card. Normally, this causes the applet to select a different DF within its file system hierarchy (if the applet has a file system).

2.9.4 process()

Any non-select APDU received causes the `process` method of the active applet to be invoked. The APDU is supplied as a parameter. The applet may respond to the APDU with data and returns from `process` with a SW. The SW and optional response data are sent to the CAD.

APDU processing is describe in more detail in a later section.

2.9.5 Applet Internal State

After installation, an applet is completely responsible for its own state and may decide how to respond to each invocation of its `select` or `process` methods.

Some smart card application specifications call for applets to "block" themselves or otherwise maintain state indicating what the applet can and cannot do at any point in time. Applets must manage this state themselves.

Any select APDU with this applet's AID will cause this applet's `select` method to be invoked. When this applet is active, any select APDU with non-recognized AID and any other APDU will cause this applet's `select` or `process` methods to be invoked.

2.9.6 Applet Processing

Once selected, an applet is active until platform power is lost or until another applet is selected. During this time, the applet receives, processes, and responds to APDU commands from the CAD. As part of this processing and applet may:

- Maintain its own state (including states like "blocked" or "expired").
- Reference (read and write) its own objects.
- Reference objects which have been appropriately shared.
- Share its objects with other applets.

- Enclose multiple updates in a transaction.
- Create new objects (if the issuer policy allows this).
- Invoke services provided by the Java Card API, such as PIN, crypto, and FileSystem.

3. APDU Handling

3.1 Overview

The Java Card APDU class provides a powerful and flexible mechanism to handle ISO 7816-4 APDUs. It is optimized so as to be efficiently implementable on small Java Card platforms.

It is also carefully designed so that the intricacies of and differences between the T=0 and T=1 protocols are hidden from the applet developer. In other words, using the APDU class, applets can be written so that they will work correctly regardless of whether the platform is using the T=0 or T=1 transport protocol.

The next section describes the methods in the APDU class in the order that they will typically appear in applet code. A later section describes how a typical applet might handle the various "cases" of APDUs.

3.2 Methods

3.2.1 process()

Even though the process method is the Applet class, it is the beginning of APDU handling. All APDU commands (except for `install` and `select`) are delivered to the active applet via its `process` method.

3.2.2 Returning from process()

At any point in time, the applet may return a SW from `process`. This terminates the processing of that APDU. The SW is returned to the CAD.

The Java Card system ensures that the underlying transport protocols are properly managed so that the CAD and card do not become unsynchronized. This may require the reading and discarding of unread command data bytes and/or the padding of incomplete responses with null bytes.

The desired result is that when the applet encounters a problem, it can simply return the desired SW from `process` and the system will take care of all other details.

3.2.3 getBuffer()

The command data bytes received and the response bytes to be sent are stored in the APDU object's buffer. `getBuffer` obtains the reference to the buffer so that the applet can examine the command bytes and store the response bytes using normal Java syntax for array access.

The size of this buffer is platform dependent. The minimum buffer size is 32 bytes. Platforms with larger RAMs will usually have a larger APDU buffer. Because the buffer is declared as a byte array, its size can be obtained using normal Java syntax (the *array*.length keyword).

The APDU buffer object belongs to the Java Card system, but it is "shared" with all applets. Applets should not store data in this buffer between invocations of `process` because the system is not guaranteed to preserve such data. Furthermore, the contents of the buffer are cleared for each select APDU so that any private data from one applet cannot be seen by another applet.

Furthermore, the APDU buffer is a transient object because its contents need not be preserved across CAD sessions and because fast update is important.

3.2.4 getIFSC()

ISO 7681-3 defines the T=1 IFSC value as "the maximum length of information field of blocks which can be received by the card." For the T=1 protocol, this value is platform dependent and is specified in the ATR.

In the APDU class, the IFSC is used in a protocol-independent way to indicate "the maximum number of bytes which can be received into the APDU buffer in a single I/O operation." The I/O operations are the `receiveBytes`, `setIncomingAndReceive`, `sendBytes`, and sendBytesLong methods.

The IFSC can be as large as the APDU buffer, but will typically be somewhat smaller. This allows an applet to preserve a few bytes of data in the APDU buffer and still receive subsequent command data bytes without the risk of overflow. For example, a platform might have an APDU buffer size of 64 bytes and an IFSC of 60 bytes.

3.2.5 Reading the APDU Header

When an applet's `process` method is invoked, the first 5 bytes of the APDU buffer contain the APDU header bytes. The remaining bytes in the buffer are undefined and should not be read or written by the applet. At this time, the applet should only examine the following values:

- Buffer[0] = CLA, the APDU class byte.
- Buffer[1] = INS, the APDU instruction byte.
- Buffer[2] = P1, the APDU parameter 1 byte.
- Buffer[3] = P2, the APDU parameter 2 byte.
- Buffer[4] = P3, the APDU fifth byte, which is:
  - For case 1, P3 = 0.
  - For case 2, P3 = Le, the length of expected response data.
  - For cases 3 and 4, P3 = Lc, the length of command data.

The applet should examine these values in order to determine what to do next.

As previously mentioned:

- The applet may return from `process` with a SW at any time, regardless of the "case" of the APDU.
- The applet need not know which protocol (T=0 or T=1) is actually being used. The Java Card system and API will handle all protocol details.

3.2.6 setIncoming()

If the applet determines (usually via INS) that the APDU has command data (case 3 or 4), it should call `setIncoming`. This informs the underlying protocol handler that the fifth byte of the header is Lc and to expect incoming command data bytes.

3.2.7 receiveBytes()

The applet receives a group of command bytes by calling `receiveBytes`, specifying the offset into the APDU buffer where the group of bytes is to be placed. In cases where IFSC is smaller than the buffer size, this allows the applet some flexibility. For example, the applet may have processed a group of data except for a few bytes. The applet can move these bytes to the beginning of the buffer, and then receive the next group such that it is appended to the bytes still in the buffer. This feature is important in case the command data is split across a group of bytes that needs to be processed as a whole.

The actual number of received bytes is returned by `receiveBytes`. Due to the operation of the T=1 protocol, the applet has no control over how many bytes are received. Typically, the number of bytes will be the minimum of IFSC and the total number of command bytes remaining to be received. However, this cannot be guaranteed. Depending on the implementation of the CAD's protocol handling, a call to `receiveBytes` could receive less than that amount.

After processing each group of command data bytes, the applet can call `receiveBytes` to get additional groups of command bytes (if any).

3.2.8 setIncomingAndReceive()

For efficiency, the `setIncomingAndReceive` method is a combination of `setIncoming` and `receiveBytes(5)`. In other words, `setIncoming` is done and then the first group of command data bytes is received at offset 5 in the buffer (so that the command header is preserved). As with `receiveBytes`, the number of command data bytes received is returned.

3.2.9 setOutgoing()

After processing incoming bytes (if any) the applet can send response bytes. If the APDU command is case 2 or 4, the applet calls `setOutgoing` to indicate that it wishes to send response data. `setOutgoing` switches the internal APDU state to "send." It also returns the Le as follows:

- For case 2, Le = P3, the fifth byte of the APDU header.
- For T=1 case 4, Le = the actual Le from the end of the command bytes.
- For T=0 case 4, Le = 256 because the actual Le cannot be determined. So the maximum Le allowed for T=0 is assumed.

Note: Even though the above text refers to the transport protocols for clarity of explanation, it is not necessary for the applet to know which protocol is being used.

3.2.10 setOutgoingLength()

After examining Le, the applet must indicate by calling `setOutgoingLength` how many **total** response data bytes (not including SW) it will actually send. The default value is 0, so this method need not be called if the applet will not be sending response bytes.

The total number of response bytes could be more than will fit in the APDU buffer. In this case, the applet will have to break the response up into groups of bytes and send one group at a time.

3.2.11 sendBytes()

The `sendBytes` method sends a group of response bytes from the APDU buffer. If the applet needs to send multiple groups, it must call this method repeatedly until all bytes are sent.

3.2.12 sendBytesLong()

The `sendBytesLong` method is similar to `sendBytes`. However, it allows the applet to send a group of bytes from any byte array (except from the APDU buffer byte array). This is useful in cases where the data to be sent is in a file or in some other data structure's byte array.

`sendBytesLong` simply copies smaller groups of bytes into the APDU buffer and sends them one at a time. For this reason, the applet should not expect the contents of the APDU buffer to be preserved after a call to `sendBytesLong`.

3.2.13 wait()

Both the T=0 and T=1 protocol have a mechanism by which the card can request additional time from the CAD, so that the protocol does not time out while the card is performing a long computation. The `wait` method invokes this mechanism. It can be called at any time during APDU processing when the applet must do something else for an extended period of time.

`wait` is intended to be implemented as follows (see ISO 7816-3 for additional details):

- For T=0, `wait` causes a NULL procedure byte (0x60) to be sent to the CAD. This resets the "work waiting time."
- For T=1, `wait` causes a "S(WTX response)" to be sent to the CAD. This requests additional block waiting time (BWT). [Note: to do - determine how much extra time the WTX should ask for.]

3.3 Cases

This section gives example of how each of the APDU cases can be handle by the Java Card APDU API.

3.3.1 Case 1 – no command data, no response data

1. The applet's `process` method is called. The applet examines the first 4 bytes of APDU buffer and determines that this is a case 1 command (P3 = 0).
2. The applet performs the request.
3. The applet returns from the `process` method with the appropriate SW.

3.3.2 Case 2 - no command data, send response data

1. The applet's `process` method is called. The applet examines the first 5 bytes of APDU buffer and determines that this is a case 2 command. Le is in P3 of the header.
2. The applet calls `setOutgoing`.
3. The applet calls `sendBytes` or `sendBytes` (repeatedly if necessary) to send groups of response bytes.
4. The applet returns from the `process` method with the appropriate SW.

3.3.3 Case 3 – receive command data, no response data

1. The applet's `process` method is called. The applet examines the first 5 bytes of APDU buffer and determines that this is a case 3 command. Lc is in P3 of the header.
2. The applet calls `setIncoming` or `setIncomingAndReceive`, followed by repeated calls (if necessary) to `receiveBytes`. Each group of command data bytes is processed as it is received.
3. The applet returns from the `process` method with the appropriate SW.

3.3.4 Case 4 – receive command data, send response data

Case 4 is simply a combination of cases 3 and 2. The applet calls `setIncoming` and handles the command bytes as described for case 3. Then the applet calls `setOutgoing` and handles the response bytes as described for case 2.

4. File System

TBD.

5. Secure Install Process

TBD.