Java Card 2.0
Programming Concepts

October 15, 1997

Revision 1.0 Final

Document Revision History

<u>Revision 1.0</u>

This document was previously called the "Java Card 2.0 Programmer's Guide". Since the information contained in this document explains the concepts behind Java Card 2.0, it has been renamed to "Java Card 2.0 Programming Concepts."

Changes include:

- Addition of a Preface.
- Additional information on all topics.
- "File System" and "Cryptography" sections added.

Contents

Preface

The Java Card 2.0 Programming Concepts manual contains information on the Java Card 2.0 classes and how they can be used in your applet or framework. The concepts in this manual include:

- An explanation of the Java Card framework
- How to implement the Java Card 2.0 classes in your applet
- Applet lifetime and runtime environment
- How applets share objects securely
- How transactional atomicity works in Java Card
- How the ISO file system is implemented in an object-oriented Java Card

Note: The process of developing, distributing, and installing applets is beyond the scope of this document.

Java Card Runtime Environment implementations, such as programming and configuring an ATR, are also beyond the scope of this document.

Who Should Use This Manual

The *Java Card 2.0 Programming Concepts* manual is targeted at developers who are creating applets using the Java Card 2.0 API and also at developers who are considering creating a vendor-specific framework based on the Java Card 2.0 API.

Java Card 2.0 assumes a particular model (ISO 7816 or EMV) and supports a framework designed for that model. This guide describes how a developer can implement the Java Card 2.0 API to write applets for smart cards using this model and framework.

For developers considering creating a vendor-specific framework, this guide provides additional information on the required behavior of the classes in the Java Card 2.0 API.

A knowledge of the Java programming language, object-oriented design, and smart cards is assumed.

References

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

- *The Java Card 2.0 API*
- *The Java Card 2.0 Language Subset and Virtual Machine*
- *The ISO 7816 Specification Parts 1-6*
- *The EMV '96 (Europay, MasterCard, Visa) Integrated Circuit Card Specifications for Payment Systems.*
- *The Java Language Specification* by James Gosling, Bill Joy, and Guy Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1.

- *The Java Virtual Machine Specification* by Tim Lindholm, and Frank Yellin. Addison-Wesley, 1996, ISBN 0-201-63452-X.

How This Manual Is Organized

This manual is organized into five main sections.

Section 1, “The Java Card Framework,” describes the Java Card framework and how it affects you as an applet developer.

Section 2, “Applet Design Concepts,” this section describes how you can implement the classes defined in the *Java Card 2.0 API* in your vendor-specific applet. This section includes programming considerations for your applet.

Section 3, “APDU Handling,” explains the methods in the ISO-compliant APDU class and also describes how a typical applet might handle the various cases of APDU's.

Section 4, “Java Card File System,” describes how an ISO-compatible file system is implemented in the object-oriented Java Card.

Section 5, “Cryptography,” describes the API for cryptography methods.

Terminology

The following terms are used throughout the manual:

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

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

Applet the basic unit of selection, context, functionality, and security in a Java Card.

Applet developer refers to a person creating an applet using the Java Card 2.0 API.

Applet execution context. The JCRE keeps track of the currently selected applet as well as the currently active applet. The currently active applet value is referred to as the *applet execution context.* When a virtual method is invoked on an object, the applet execution context is changed to correspond to the applet that owns that object. When that method returns, the previous context is restored. Invocations of static methods have no effect on the applet execution context. The applet execution context and sharing status of an object together determine if access to an object is permissible.

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

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

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

Java Card Runtime Environment (JCRE) consists of the Java Card Virtual Machine and the core classes in the Java Card API.

JCRE Implementer refers to a person creating a vendor-specific framework using the Java Card 2.0 API.

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

Transaction is an atomic operation where the programmer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

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

1. The Java Card Framework

Java Card 2.0 assumes a particular model (ISO 7816 or EMV) and supports a framework designed for that model. The following section describes this Java Card framework.

Note: Java Card 2.0 does not preclude vendors from including on their cards additional frameworks and /or applications designed for different models as long as the standard framework is supported. In such cases, however, it is the vendor's responsibility to solve the issues related to the coexistence of different frameworks.

ISO 7816 contains standards for ICCs (Integrated Circuit Cards with contacts), also known as smart cards. These standards cover the various aspects of smart cards, such as:

- Physical characteristics (Part 1)
- Dimensions and location of the contacts (Part 2)
- Electronic signals (Part 3, initial sections)
- Transmission protocols (Part 3, later sections)
- Inter-industry commands for interchange (Part 4)
- Application identifiers (Part 5)
- Inter-industry data elements (Part 6)
- Inter-industry commands for SCQL (Part 7)

Smart cards can be used in a wide variety of applications. Since most smart cards have very limited hardware resources, few smart card systems are designed to support all the features of all the parts of ISO 7816. Typically, vendors of smart card implementations and applications will identify a subset of ISO 7816 with which to be compatible. They may also support proprietary features as required for their targeted industries.

For example, a smart card system might be compatible with Parts 1 and 2 and half of Part 3 (Electronic Signals) but might require proprietary transmission protocols and commands, and so not be compatible with the latter half of Part 3 or the remaining Parts.

The EMV[1] standard, defined by members of the international financial community, combines a subset of ISO 7816 Parts 1-6 with additional proprietary features into a design tailored to meet the specific needs of their industry.

The Java Card 2.0 API is designed to easily support those smart card systems and applications which are generally compatible with ISO 7816 Parts 1-6 and/or EMV. Java Card 2.0 defines a framework within which applications can be written. The framework automatically takes care of most of the low-level details specified in ISO 7816 Parts 1-3. The framework also provides classes and methods that assist applications in being compatible with Parts 4-6 and/or EMV.

[1] EMV '96 (Europay, MasterCard, Visa) Integrated Circuit Card Specifications for Payment Systems.

The major advantage of this framework is that it is relatively easy to create an applet that fits into the model assumed by the framework. The applet developer can concentrate most of his/her effort on the details of the application, rather than on the details of the smart card system infrastructure.

The rest of this document explains how an applet developer can implement the Java Card 2.0 API using the model and framework described above. The concepts described in this manual can also be used by the JCRE implementer in creating a compatible implementation.

2. Applet Design Concepts

According to ISO 7816-4, an ICC (Integrated Circuit Card with contacts), or smart card, contains one or more applications. In place of the ISO term *application*, Java Card uses the term *applet*.

Applets are the basic unit of selection, context, functionality, and security. When a smart card is inserted into a CAD, the CAD selects an applet on the card and sends 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 indicate result of the operation using a status word (SW) as defined in ISO 7816-4.

Applets are objects that are instances of classes defining the specific behavior of that applet. Applet classes are subclasses of the `javacard.system.Applet` class, which defines the common behavior of all applets.

2.1 Multiple Applets

A CAD can interact with a single applet. Or it can interact with several applets by selecting each in turn and sending them individual APDU commands to perform. For example, first the CAD selects applet1. After it completes a transaction and is finished with applet1, the CAD then selects applet2 and processes transactions using applet2.

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 the other applets on the card. However, Java Card provides facilities to support more sophisticated scenarios in which multiple applets can discover each other, communicate, and share data in a limited manner, while still maintaining protection from each other in the form of a firewall between applets.

2.2 Packages

As in standard Java, Java Card supports packages. Packages, like applets, are named with Application Identifiers. You, the applet developer, do not assign an AID in your applet or package source code. The AIDs are associated with your applet and packages when they are prepared for installation onto the card. This process is beyond the scope of this document.

An applet can be delivered as one or more packages that are installed on the card and linked to each other and to other packages already on the card. A minimal applet is a package with a single class derived from the `javacard.framework.Applet` class. A package containing the applet class must be linked with the framework package. Packages not containing an applet class can also be loaded onto the card.

The AID of the package must be different than the AID of the applet.

As an applet developer, you can create a package with one applet. This package can then be installed onto the card. Or you can create a package that contains classes that can be shared by multiple applets.

2.3 Objects

In Java Card, applets are written in the Java language and thus use Java objects to represent, store, and manipulate data.

Every object (class instance or array) on the card is owned by the applet which instantiated it. The owning applet always has full privileges to use and modify the object. Unless an object is explicitly shared, it is only accessible to the applet that created it. The standard Java rules apply:

- An object continues to exist as long as a valid reference to that object exists on the stack, in a local or parameter variable, or in a field of another existing object.
- All fields in a `new` object are set to their default values.

Because smart card resources are limited, you, the JCRE implementer, are free to decide whether or not to implement garbage collection of persistent objects. However, if garbage collection is not implemented, then you must be careful when allocating “temporary” objects so as to avoid wasting memory.

2.4 Lifetime of the Virtual Machine

In a PC or Workstation, the Java Virtual Machine runs as an operating system process. When the OS process is terminated the Java applications and their objects are automatically destroyed.

By contrast, in Java Card the execution lifetime of the Virtual Machine is the lifetime of the card. Most of the information stored on a card must be preserved even when power is removed from the card. Persistent memory technology (such as EEPROM) enables a smart card to store information even when power is removed. Since the Virtual Machine and the objects created on the card are used to represent application information that is persistent, the Java Card Virtual Machine runs forever.

The JCRE is initialized and the framework is created and initialized at card initialization. The framework exists for the lifetime of the Virtual Machine. Because the execution lifetime of the Virtual Machine and the framework span CAD sessions of the card, the lifetimes of objects created by applets will also span CAD sessions. Objects that have this property are called persistent objects. The JCRE implementer must make an object persistent when:

- The `Applet.register` method is called the JCRE stores a reference to the instance of the applet object. The JCRE implementer must ensure that instances of class applet object are persistent.
- A reference to an object is stored in a field of any other persistent object. This requirement stems from the need to preserve the integrity of the JCRE's internal data structures.

If an object is not referenced by other persistent objects, it can be discarded or garbage collected.

2.5 Transient Objects

Applets sometimes have objects that contain temporary, or transient, data that need not be persistent across CAD sessions. Java Card supports designating objects as transient.

Transient objects are different from persistent objects in the following ways:

- The contents of the fields of a transient object are reset by the JCRE to the default state (zero, null, or false) during power loss or termination of CAD sessions. Transient object's values will not persist from one CAD session to the next and shorter lifetimes are possible.
- Updates to the fields of a transient object are not transacted. Atomicity does not apply to transient objects. If the contents of a transient object are changed during a transaction, the new value is permanent. If the transaction is aborted, the modified value remains.

These properties make transient objects ideal for small amounts of applet temporary data that are frequently modified but that need not be preserved across sessions or shorter time periods.

A transient object can be persistent. But the contents of the object's fields are not. This means that other objects can have references to transient objects, and those references can be stored persistently.

You can make an object transient using the API call:

```
System.makeTransient(Object anObject, byte duration)
```

The duration may be one of the following values:

`System.TRANSIENT_SESSION` - contents of the object are reset at the end of each CAD session, or when the card is removed from the CAD.

`System.TRANSIENT_SELECTION` - contents of the object are reset when the object's owning applet is deselected (or some other applet is selected).

`System.TRANSIENT_APDU` - contents of the object are reset when the method `Applet.process()` returns.

When you use `makeTransient` to make an object transient, all the object's fields are reset to their default values (zero, null, or false). In addition, every time the card is reset, all the fields of all transient objects are reset to their default values. Transient objects with a shorter duration will also have their contents reset to default values at the specified times.

You can only make an object transient once. After `makeTransient` has been called with an object once, any successive calls with that object will throw a `SystemException` with a reason code of `ALREADY_TRANSIENT`.

You can determine if an object is transient by using the following API call in your applet:

```
byte System.isTransient(Object anObject)
```

This method will return one of the above duration constants, or the constant `System.TRANSIENT_NONE` to indicate the object is not transient.

2.6 Atomicity

Atomicity defines how the card handles the contents of persistent storage after a stop, failure, or fatal exception during an update of a single object or class field. If power is lost during the update of a field in a persistent object, the applet developer must know what the field contains when power is restored.

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

Your applet might not require atomicity for array updates. The `Util.arrayCopyNonAtomic` method is provided for this purpose.

2.6.1 Transactions

Your applet might need to atomically update several different fields in several different objects. That is, either all updates take place correctly and consistently or else all fields are restored to their previous values.

Java Card supports a transactional model in which an applet can designate the beginning of an atomic set of updates with a call to the method `System.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 `System.commitTransaction`, all conditional updates are committed to persistent storage.

If power is lost or if some other system failure occurs prior to the completion of `System.commitTransaction`, 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 `System.abortTransaction`.

2.6.2 Nested Transactions

The model currently assumes that nested transactions are not possible. There can be only one transaction in progress at a time. If `System.beginTransaction` is called while a transaction is already in progress, then a `TransactionException` is thrown.

The `System.transactionDepth` method is provided to allow you to determine if a transaction is in progress.

2.6.3 Transaction Failure

If power is lost or the card is reset or some other system failure occurs while a transaction is in progress, then all conditionally updated fields since the original `System.beginTransaction` are restored to their previous values.

This action is performed automatically by the JCRE when it reinitializes the card after recovering from the power loss, reset, or failure. The JCRE determines those objects (if any) which were conditionally updated and restores them. For example, a JCRE implementer might choose to do this during card initialization before the ATR is sent to the CAD.

2.6.4 Aborting a Transaction

If the applet encounters an internal problem or decides to cancel the transaction, you can programmatically undo conditional updates by calling `System.abortTransaction`. If this method is called, all conditionally updated fields since the original `System.beginTransaction` are restored to their previous values and the `System.transactionDepth` value is reset to 0.

2.6.5 Transaction Duration

If an applet returns from the methods `select`, `deselect`, or `process` with a transaction in progress, the JCRE will automatically abort the transaction.

2.6.6 Transient Objects

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.”

2.6.7 Commit Capacity

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 exception is thrown if the commit capacity is exceeded.

2.6.8 TransactionException

A `TransactionException` a subclass of `RuntimeException` is thrown when certain kinds of problems are detected within the transaction subsystem. The possible reason codes are as follows:

1. TRANS_IN_PROGRESS — `beginTransaction` was called while a transaction was already in progress. This is a programming error.
2. TRANS_NOT_IN_PROGRESS — `commitTransaction` or `abortTransaction` was called while a transaction was not in progress. This is a programming error.
3. TRANS_BUFFER_FULL — During a transaction, an update to persistent memory was attempted which would have caused the commit buffer to overflow. The update was not performed and the transaction is still in progress (it is neither aborted nor committed). This is generally a fatal programming error. You must alter the code to use shorter transactions that do not cause the commit buffer to overflow.
4. TRANS_INTERNAL_FAILURE — Some kind of internal fatal problem occurred within the transaction subsystem. This exception indicates that there is a problem in the transaction subsystem.

2.7 Applet Isolation and Object Sharing

To create a secure and trusted environment, applets are isolated from each other. An *applet firewall* prevents one applet from accessing the contents or behavior of objects owned by other applets.

Every object (class instance or array) on the card is owned by the applet which instantiated it, that is, the applet which was active at the time the object was created. The owning applet always has full privileges to use and modify the object.

The applet firewall ensures that no other applet may use, access, or modify the contents of an object owned by another applet except as described in this section. This does not restrict another applet from having a reference to such an object, but that applet cannot invoke methods on the object or get or set its field contents.

However, it is necessary to allow exceptions to this restriction. The JCRE must be able to invoke methods on applets, applets must be able to use objects owned by the JCRE, and applets must be able to interoperate for cooperative applications such as loyalty programs.

If an applet does not have sharing privileges for an object, any attempt to invoke an instance method or access the object's contents will throw a `SecurityException`. This exception is thrown on illegal attempts to invoke a virtual method, access a field value, check an array length, cast the object to a different type, or use the `instanceof` operator. The JCRE implementation can choose to mute the card instead of throwing the exception.

When the JCRE checks an object's owner against the current applet execution context, it allows the exceptions listed in the following sections to pass the check. For most operations, there is no special action needed. For method invocation operations, the JCRE remembers the old context, and performs an applet context switch to allow the code in the object's applet to function correctly and with expected security restrictions. When the method returns, the old applet context is restored.

2.7.1 JCRE Privileges

The JCRE is allowed to use and modify any object on the card. For instance, the JCRE might invoke methods on applet instances without the instances having to be shared.

2.7.2 Unrestricted Sharing

An applet may permit *unrestricted* sharing of any of its objects. The applet shares an object by making the call:

```
System.share(anObject);
```

Note: Only the applet that owns the object may make this call. Sharing privileges may not be revoked. That is, once an object is shared, it is shared for its remaining lifetime.

2.7.3 Restricted Sharing

An applet may permit *restricted* sharing of any of its objects by making the call:

```
System.share(anObject, AID);
```

Note: Only the applet that owns the object may make this call. Sharing privileges may not be revoked. That is, once an object is shared, it is shared for its remaining lifetime.

An applet can make a restricted sharing call multiple times with the same object to allow the object to be shared with multiple applets, but not with all applets.

2.8 Exception Handling

Java Card supports exception handling as defined in the Java language. Exceptions are thrown by the Virtual Machine 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 standard Java way.

Checked exceptions (see *The Java™ Language Specification*) are subclasses of `Exception` 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 Java Card API.

Unchecked exceptions are subclasses of `RuntimeException` and need not be caught nor declared in a `throws` clause. These exceptions are typically indicative of unexpected runtime problems, programming errors or error processing states, and are caught by the outmost levels of the JCRE. However, you can have your applet catch unchecked exceptions if you choose to do so.

`ISOException` is a special unchecked exception. It is raised during runtime to denote the warning or error processing state in the card. `ISOException` holds a status word (SW). The status word values are defined in the ISO 7816-4 specification. When an applet or the underlying JCRE code encounters a problem and decides to terminate the process, it simply throws an `ISOException` with the appropriate status word. The JCRE catches an `ISOException` and returns the SW as a part of the Response APDU to the CAD.

An exception that is not caught by an applet is caught by the JCRE. All exceptions other than `ISOException` indicate a fatal error in a card. The JCRE implementer may handle these fatal errors individually in one of the following ways:

1. Reply to the current APDU command (if any) with a Status Word of `ISO.SW_UNKNOWN` (0x6F00).
2. Put the card into a mute state. A muted card will not respond to any APDU commands, including the currently executing command.

Option 1 is preferred for debugging environments. Since an uncaught exception might indicate a security attack on the card, option 2 may be preferred for production cards.

2.8.1 Exception Objects

Since garbage collection is not required by the Java Card standard, when a Java Card exception object is created, it may continue to occupy precious memory space, even if there is no longer a reference to it. To optimize your memory usage, you can 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, you can have your code retrieve and reuse the reference for the desired exception object from the well-known location, fill in the reason code in the object, and `throw` the object.

The JCRE pre-creates an instance of each kind of specific exception defined in the Java Card API. Most of these are unchecked exceptions. When these exception objects are needed, use the static method `throwIt.`

You can define your own exceptions by creating subclasses of class `Exception`. These are always checked exceptions. These exceptions can be thrown and caught as desired by the applet. However, during initialization you should have your applet 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.

2.9 Applet Lifetime and Runtime Environment

Because applet objects exist for the life of the card, once installed an applet lives on the card forever. Each applet is a subclass of the `Applet` class. As defined by this template, the JCRE interacts with the applet via its public methods `install`, `select`, `deselect` and `process.` Your applet must implement the `install method`. If the `install` method is not implemented, the applet's objects cannot be created or initialized.

For the purposes of this document, an applet's lifetime begins at the point where it has been correctly loaded into card memory, linked, and otherwise prepared for execution. The last phase of this installation process is when the JCRE calls the `install` static method of the `Applet's` 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.

It is not necessary to set up a transaction in the `install` method, as the JCRE will ensure that the `install` method is called from within a `beginTransaction / endTransaction` block.

Typically, an applet will create various objects, initialize them with predefined values, set some internal state variables, and call the `Applet.register` method to inform the JCRE that the applet is available for selection.

If the applet encounters a problem with installation, it may throw an `ISOException` with the appropriate status word. The JCRE will abort the applet installation and return the status word to the CAD.

Simple applets might 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. The applet developer is responsible for setting internal state variables in his/her applet to track these state transitions.

2.9.2 select()

Applets remain in a suspended state until they are explicitly selected. Selection occurs when the JCRE receives a SELECT APDU in which the name data matches the AID of the applet. Selection causes an applet to become active, and the applet execution context is adjusted so that only objects belonging to this applet (or appropriately shared to this applet) can be accessed.

The JCRE informs the applet of selection by invoking its `select` method. The applet may decline to be selected by returning `false` from the call to the `select` method or by throwing an exception. If the applet returns `true`, the actual select APDU command is supplied to the applet in the subsequent call to its `process` method, so that the applet may respond with selection response data such as an FCI, as defined by ISO 7816-4.

The actual select APDU command is supplied as a `select` parameter, so that your applet can examine the APDU contents. The applet may respond to the select APDU with data (see the `process` method for details) and flag errors within `select` by throwing an `ISOException` with the appropriate SW. The SW and optional response data are sent to the CAD. If the applet throws an exception, it indicates that the applet cannot be selected, and the applet will no longer be selected.

A JCRE implementer might want to support a feature by which a default applet is always selected when the card is reset. If so, then the applet's `select` method must still be called.

After successful selection, all subsequent APDUs, including the original select APDU, are delivered to this applet via the `process` method. If a select APDU contains the name of another applet, or even this same applet, the previously selected applet becomes deselected. The JCRE indicates the condition of becoming inactive to the applet by invoking its `deselect` method. The newly identified applet then becomes active and its `select` method is called.

If a select APDU contains a name that is not recognized by the JCRE as the AID of an applet, then the `process` method of the active applet is called. Normally, this should cause the applet to select a different DF within its file system hierarchy (if the applet has a file system).

> ***Note***: Although the Java Card framework is designed to support applets which are generally compatible with EMV and/or ISO 7816 Parts 1-6, other kinds of applets supported by other kinds of frameworks may co-exist on the card. When this is the case, it is the responsibility of the JCRE implementer to define how selection works for applets that are not identified by an AID.

2.9.3 process()

Any APDU received causes the `process` method of the active applet to be invoked. The APDU is supplied as a parameter. The applet may optionally respond to the APDU with data. On normal return, the JCRE automatically appends 0x9000 as the completion response SW.

At any time during `process`, the applet may throw an `ISOException` with an appropriate SW. This optional response data and SW are sent to the CAD.

APDU processing is described in more detail in the "APDU Handling" section.

2.9.4 deselect()

Whenever a select APDU command causes an applet to become deselected, the JCRE calls the `deselect` method of that applet. This allows the applet to perform any cleanup operations that may be required to allow some other applet to execute.

Upon `select`, your applet implementation might need to know whether a card reset has occurred. The `deselect` method can be used to track the difference. Note that the `deselect` is never invoked upon reset.

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`, `deselect`, 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. Your 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 selected, and another select APDU containing an AID that matches that of an installed applet is received, the JCRE will invoke this applet's `deselect` method. Any other APDU will cause this applet's `process` method to be invoked.

2.9.6 Applet Processing

Once selected, an applet is selected until platform power is lost, the card is reset, 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 an 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.

2.9.7 Power loss

Power loss occurs when the card is withdrawn from the CAD or if there is some other mechanical or electrical failure. When power is reapplied to the card the JCRE ensures that:

- All transient object fields are reset to their default state.
- The transaction in progress, if any, is aborted.
- The applet becomes deselected. Note that the `deselect` method is not called.

3. APDU Handling

The Java Card APDU class provides a powerful and flexible mechanism to handle APDUs whose command and response structure conform to the ISO 7816-4 specification. The APDU class is optimized for 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.

3.1 Methods

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

3.1.1 process()

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

For security reasons, the unused bytes in the APDU buffer are cleared to zero prior to invoking the applet's `process` method.

3.1.2 Returning error response in process()

At any point in time, the applet may throw an `ISOException` with the appropriate status word (SW) in `process`. The SW contained in `ISOException` object is returned to the CAD.

The JCRE ensures that the underlying transport protocols are properly managed so that the CAD and card do not become unsynchronized. This might require the reading and discarding of unread command data bytes. If the JCRE is implemented in this manner, when the applet encounters a problem, it can simply throw an `ISOException` with the SW code and the JCRE will take care of all other details.

On normal return the JCRE will send the normal completion status bytes (0x9000) to the CAD on behalf of the applet.

3.1.3 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 37 bytes (5 bytes of header plus the default size of IFSC (as defined by ISO 7816-3)). Platforms with larger RAM capacity 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 APDU buffer object belongs to the JCRE, but it is shared with all applets. Applets should not store data in this buffer between invocations of `process` because the JCRE 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.

To reduce protocol overhead, the `setOutgoingAndSend` method can rely on the buffer being unaltered after its invocation. Again, to reduce protocol overhead, when `sendBytes` or `sendBytesLong` is called to send the last of the response, you, the JCRE implementer, can choose to transmit the data at a later time. The buffer must not be altered after the last send invocation.

3.1.4 getInBlockSize()

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.

T=0 protocol has no such maximum length requirements and need only receive 1 byte at a time. Thus, this method returns 1 if the underlying protocol is T=0.

In the APDU class, the InBlockSize 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` methods.

The InBlockSize plus the header bytes (5) 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 69 bytes and an InBlockSize of 64 bytes.

3.1.5 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.

You should have your applet examine these values in order to determine what to do next.

Remember that the applet may throw an `ISOException` with the appropriate SW in `process` at any time, regardless of the "case" of the APDU, and The applet need not know which protocol (T=0 or T=1) is actually being used. The JCRE and API will handle all protocol details.

3.1.6 setIncomingAndReceive()

If the applet determines (usually via INS) that the APDU has command data (case 3 or 4), it should call `setIncomingAndReceive`. This informs the underlying protocol handler that the fifth byte of the header is Lc and to receive a group of incoming command data bytes starting at offset 5 in the buffer. The actual number of received data bytes is returned by `setIncomingAndReceive.`

3.1.7 receiveBytes()

After processing each group of incoming command data bytes, the applet can get additional groups of command bytes (if any) by calling `receiveBytes` specifying the offset into the APDU buffer where the group of bytes is to be placed. This allows the applet to control how the APDU buffer is used in the processing of incoming data. 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 instances where some 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`. Because of 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 InBlockSize 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.

3.1.8 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.

3.1.9 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.

If the underlying protocol is T=0, and the *total* response data bytes to be sent is not exactly equal to Le, this method will prompt the terminal for a GET RESPONSE command with the indicated *total* response length. This ensures that the applet need not do any special processing based on the underlying protocol.

The total number of response bytes might be too large to fit in the APDU buffer. In this case, you must have the applet break the response up into groups of bytes and send one group at a time.

Tip: If the data is in a byte array, this can be done most conveniently using the `sendBytesLong` method.

3.1.10 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.

To reduce protocol overhead, the implementation may choose to defer the transmitting of the data in the last `sendBytes` until return from `process`. Therefore, the buffer must not be altered after the last `sendBytes` invocation.

3.1.11 setOutgoingAndSend()

When the entire response fits within the buffer, the setOutgoingAndSend method can be very useful. For efficiency, it combines `setOutgoing, setOutgoingLength, and sendBytes` into one call. In addition, since the entire message fits in the buffer, the response bytes and the status bytes can be sent at the same time to reduce protocol overhead. If you use this method, you must make sure that the entire message fits in the buffer and then the buffer must not be modified after this call.

3.1.12 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`.

3.1.13 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. This is invoked by the `wait` method, which can be called at any time during APDU processing when the applet must do something else for an extended period of time.

`wait` is necessary because many smart cards do not have hardware timers and/or do not support multithreaded code. In these cases, only the applet itself knows for sure how much processing is required for each command, and whether that amount of processing is likely to exceed CAD timeout values.

`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 an S(WTX request) to be sent to the CAD to request the same T=0 work waiting time quantum. This S-block requests additional block waiting time (BWT) units (see ISO 7816-3 for details).

3.1.14 getNAD()

The T=1 protocol supports the idea of context which allows an application to maintain logically multiple channels of communication with the terminal simultaneously. To allow for that possibility, the applet can use `getNAD` method to access the ISO 7816-3 defined node address and switch internal contexts if applicable. If the underlying protocol is T=0, `getNAD` will return 0.

The applet developer need not know what transport layer is in use, since node address switching in T=1 can only be initiated by the terminal.

3.1.15 getInBytesRemaining()

The `getInBytesRemaining` method is designed to be used in the `select` and `install` methods of Applet. In these Applet methods, a preread APDU is presented to the Applet. This method provides the Applet access to information about how many unread bytes remain to be received from the APDU. By subtracting this count from the data in the header Lc field (buffer[4]) the Applet can determine the number of pre-read data bytes available in the APDU buffer.

3.2 APDU Cases

This section gives an example of how each of the APDU cases can be handled by the Java Card APDU API. In all cases, the Applet can throw an `ISOException` with the appropriate error status bytes to flag errors.

3.2.1 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.

3.2.2 Case 2 - No command data, send response data

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. The response can be short or long and is handled differently based on the size.

3.2.2.1 Short Response (entire response fits in buffer)

1. If it is a short response, the applet calls `setOutgoingAndSend`.
2. The applet returns from the `process` method.

3.2.2.2 Long Response (entire response does not fit in buffer)

1. If it is a long response, the applet calls `setOutgoing` and obtains Le.
2. If the response length is greater than Le, an error is flagged. Otherwise, the applet calls `setOutgoingLength`.
3. The applet calls `sendBytes` or `sendBytesLong` (repeatedly if necessary) to send groups of response bytes.
4. The applet returns from the `process` method.

3.2.3 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 `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.

3.2.4 Case 4 – Receive command data, send response data

Case 4 is simply a combination of cases 3 and 2. First, the handles the command bytes as described for case 3. Then the applet handles the response bytes as described for case 2.

4. File System

The Java Card File System is a standard extension to the Java Card framework classes. It is not necessary for your applet to have a file system. Even if an applet accepts APDU commands like READ RECORD, you might have the applet contain its own private data structures for records, and respond to the READ RECORD command in any way that you choose.

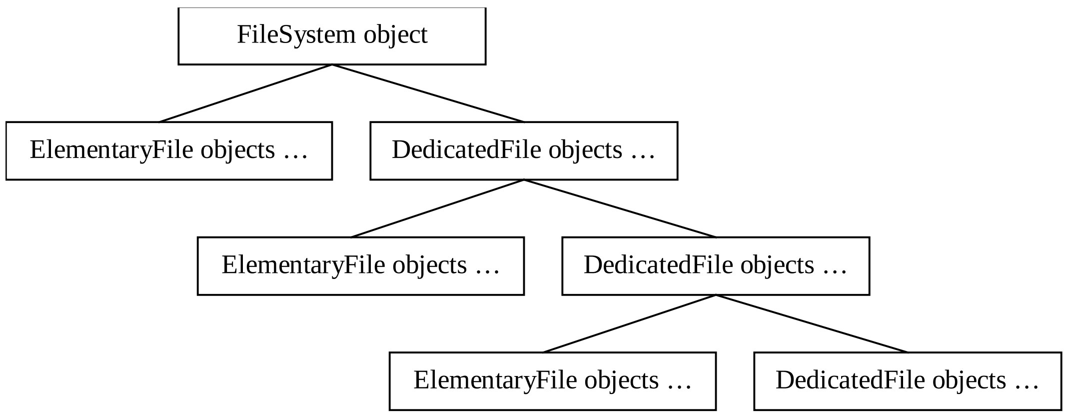
However, since so many APDU commands assume a file system structure, Java Card 2.0 provides classes to allow you to easily implement an ISO-compatible file system in the applet.

The Java Card File System provides an object-oriented design for an ISO 7816-4 compatible file system. There are two ways to view the File System organization:

1. By file system structure – the way the objects are organized into a hierarchical file system for a typical applet.
2. By class structure – the way the classes are organized into an inheritance tree in the Java language.

4.1 File System Structure

In the file system structure, objects are organized into a hierarchical file system.



The root of an applet's file system is a `FileSystem` object. Under this parent object are zero or more child objects called files. There are two kinds of file objects:

1. `ElementaryFile` (EF) objects – contain only data.
2. `DedicatedFile` (DF) objects – contain other file objects (DFs and/or EFs)

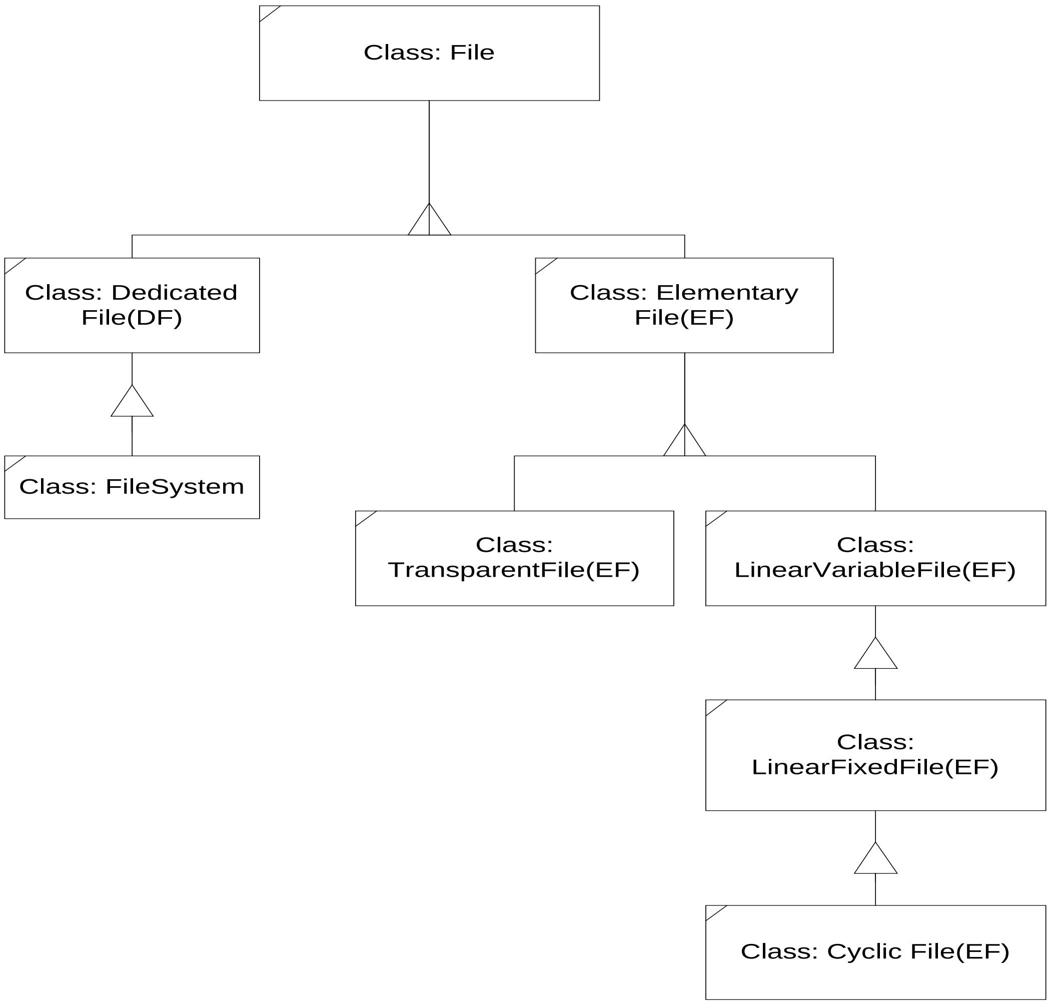
A `FileSystem` object is a special kind of `DedicatedFile` that, as the root, has additional features that ordinary DFs do not have. Normally, an applet will have one `FileSystem` (although there is nothing preventing an applet from having multiple `FileSystem` objects).

A `DedicatedFile` can contain zero or more child objects.

A `DedicatedFile` or an `ElementaryFile` can be an orphan (with no parent) or have at most exactly one parent. Therefore, a DF or an EF can be either under one `FileSystem` or not attached to any `FileSystem`.

4.2 Class Structure

This is the inheritance tree in the Java language using OMT notation.



The File class is the superclass of all classes in file system. It is an abstract class and thus can not be instantiated.

4.3 Elementary Files

Data is stored mainly in elementary files. An elementary file is organized as either a transparent data structure or as a record data structure.

- Transparent structure – The file is seen at the interface as a sequence of data units. The data in the file can be referenced by an offset.

- Record structure – The file is seen at the interface as a sequence of individually identifiable records. The records in the file can be referenced by record number or by record identifier.

Record structured elementary files can be further categorized as:

- Linear variable – Records are organized as an ordered sequence of records with various record sizes. The records in the file are kept in the order they were inserted, that is, the first inserted record is the record number one.
- Linear fixed – Similar to Linear variable, but all records are of the same length.
- Cyclic – Records are organized as a ring (cyclic structure), with fixed and equal record size. The number of records in a CyclicFile is assigned at the file creation time and can not be changed. Records are in the reverse order as they were inserted into the file. Thus the last inserted record is identified as record number one. Once the file is full, the next append instruction overwrites the oldest record in the file and it becomes the record number one.

The following diagram shows the record order of a cyclic file before an append command:

RecNum 4	RecNum 3	RecNum 2	RecNum 1

After the append record command, the new record overwrites the oldest record which is RecNum4 before the append. And the newly inserted recorded becomes RecNum1.

RecNum 1	RecNum 4	RecNum 3	RecNum 2

4.4 Dedicated Files

Dedicated files are directories in the hierarchical file structure in an applet. A dedicated file keeps references to its child objects and provides methods to search for particular DFs or EFs.

4.4.1 Class FileSystem

Class `FileSystem` is a subclass of `DedicatedFile`, and it is the root DF within an applet. Besides inheriting all the features from `DedicatedFile`, Filesystem defines additional methods to maintain internal state values, handle file-oriented APDUs, and provide file's selection and search mechanism.

- Internal state values**:** `FileSystem` maintains pointers to the current selected `DedicatedFile, ElementaryFile` and current record as well as two authorization flags as described below in the section "File Access Control." These state values are initialized as:

- currentDedicateFile = this (`FileSystem` object itself)
- currentElementaryFile = null
- currentRecordNumber = 0 (has no meaning under the context)
- authorizationFlags = false

  Current DF, EF and record are updated through their set value methods and explicit and implicit file selection commands as defined in ISO 7816.

- File-oriented APDU handling: `FileSystem` defines protected methods to handle ISO 7816-4 specified APDUs. Those methods are interfaced with applets through the `FileSystem` public method `process`. The `process` method is passed an APDU object as a parameter, and based on the instruction byte in the APDU, it dispatches the APDU handling to the following methods.
  - `Select` method is used to select an EF or DF.
  - `readBinary`, `writeBinary`, `updateBinary` and `eraseBinary` methods are used to access transparent structured files.
  - `readRecord`, `writeRecord`, `appendRecord` and `updateRecord` are used to access record structured files.
  - `getData` and `putData` are used to access data objects within the current application specific environment.

  In the applet's `process` method, you may choose first to call the FileSystem's `process` method to handle ISO file-related APDUs. There is nothing to prevent you from working around the FileSystem and allow the applet to handle file-oriented APDUs, and, therefore, directly access the specific type of file object. However, security checks are only enforced through the FileSystem interface.

- File selection: Class `FileSystem` provides public methods to allow an applet to explicitly find or select a file by FID, SFI or name. Selecting a file updates the current `DedicatedFile`, `ElementaryFile` or record in `FileSystem`. For example, If the current DF is updated, the current EF and current record number are reset to null and 0 respectively. If the current EF is updated, the current record number is reset to 0 and the current DF points to the parent of the current EF. If the current EF is null or a `TransparentFile`, the current record has no meaning in the context, and its value is set to 0. Finding a file searches through the file system and returns the appropriate file reference.

Calling the new method on the specific file subclass creates a file object. The object can then be inserted into the `FileSystem` using the `addChild` method in the `DedicatedFile`.

Refer to the *Java Card API* document for more detailed explanation on each file related class defined in `javacardx.framework`.

4.5 File Referencing

Any file (including directory files) can be referenced by a 16-bit file identifier (FID).

An elementary file can also be referenced by a short EF identifier (SFI) coded on 5 bits valued in the range from 1 to 30. For simplicity, a SFI is defined as the lowest 5 bits in the FID.

A dedicated file can possess a name with 1 to 16 bytes. When no name is provided, this field is null. To avoid ambiguously referencing a file, the following rules are defined:

- Rule 1: All EFs and DFs (Dedicated Files) immediately under a given DF should have a different file identifier.
- Rule 2: Each DF name should be unique under a given DF.

4.6 File Access Control

Since an explicit file access security model is not defined in ISO 7816-4, the Java Card 2.0 API provides a simple yet extensible scheme**.** Each file has two attributes; one for external read access and one for external write access. For each attribute, you can set one of the ALLOW_xxx values to specify what conditions must be true in order to allow that type of access (see the table below).

Table – Allow Types

Constant	**Description**
ALLOW_ANY	Any external access allowed
ALLOW_AUTH1	External access allowed only if Auth1 flag is true
ALLOW_AUTH2	External access allowed only if Auth2 flag is true
ALLOW_NONE	No external access allowed

For example, ALLOW_ANY for the read attribute means that this file can be read externally at any time. ALLOW_NONE for the write attribute means that this file can never be written externally.

By default, the read and write attributes in a file are set to “no external access allowed.”

The two Auth flags are defined in the class `FileSystem` and allow for a certain amount of applet customization. When a security attribute is set to ALLOW_AUTH1 or ALLOW_AUTH2, the access is allowed only if the appropriate Auth flag maintained by the FileSystem is true. For example, an applet may set Auth1 when a valid PIN is presented. After that point, all files with ALLOW_AUTH1 in the read attribute can now be read externally.

Note that this security checking is done in class `FileSystem`’s File-oriented APDU handling routines and is not enforced by the Virtual Machine. That is, the `FileSystem.readRecord` method will perform read access checking on the accessed file. But internal applet access to a data or directory file is not checked unless your applet specifically does so using the `isAllowed` method in the class `File`.

4.7 Interface with Native OS File System

To support JCRE implementers' native Operating System (OS) file systems, you may choose not to include any of the file system related classes defined in `javacardx.framework`. In such a case, portability might be compromised since applet developers must then dispatch to JCRE implementer specific native OS file system methods within the `Applet.process` method.

5. Cryptography

The packages `javacardx.crypto` and `javacardx.cryptoEnc` support cryptographic functionality. The `javacardx.crypto` package contains classes for performing standard cryptographic operations required in smart cards. Support for both symmetric and asymmetric cryptography is provided in a class structure that is organized to allow for easy extension and to support the subsetting required to meet import/export restrictions.

5.1 Class Structure

The Java Card cryptographic classes provide methods for:

- Encrypting and decrypting data.
- Signing data and verifying signed data
- Computing a message digest
- Generating (pseudo) random numbers

The `Key` class serves as a base for symmetric and asymmetric algorithms.

5.1.1 Symmetric Cryptography

The `SymKey` class is an abstract class defining symmetric key decryption in both ECB and CBC modes and Message Authentication Code (MAC) generation and verification. MAC operations are performed in CBC mode. Two subclasses of `SymKey` are provided: `DES_Key` (single DES) and `DES3_Key` (triple DES).

The subclasses `DES_EncKey` and `DES3_EncKey` are in the `javacardx.cryptoEnc` package; these add encryption implementations. For example, the `SymKey` class contains methods for encryption, decryption, and MAC-ing; its encryption method, however, throws a `CryptoException` with the reason `ENC_NOT_SUPPORTED`. Hence, a class like `DES3_Key`, which is intended for export, implements all of these methods except the encryption method. The class `DES3_EncKey`, not intended for export, extends `DES3_Key` and provides a full implementation the encryption method.

5.1.2 Asymmetric Cryptography

The `AsymKey` class is the base for asymmetric algorithms. The subclass `Private_Key` serves as the base class for classes supporting verification. The subclass `Public_Key` serves as the base class for classes supporting signature generation. Separate subclasses of `Private_Key` are provided for implementations of the RSA algorithm in standard modulus/exponent form and in chinese remainder theorem form.

Java Card 2.0 does not support encryption using asymmetric cryptography but the class structure is organized so that it is easily extended.

5.1.3 Message Digest

The `MessageDigest` class provides a base class for hashing algorithms. The subclass `Sha1MessageDigest` supports the SHA1 algorithm assuming a block size of 64 bytes and computing a hash value of 20 bytes.

5.1.4 Random Number Generation

The `RandomData` class is a pseudo-random number generator for use in challenge/response protocols. The quality of the randomness depends on the implementation.