# Java Card FAQ

Last revised on 10-août-01


http://www.netattitude.fr/


http://www.jguru.com/



 

1. INTRODUCTION 5

1.1 SCOPE 5
1.2 MAINTAINER 5

2. JAVA CARD 6

2.1. WHAT IS JAVA CARD? 6
2.2. WHY JAVA CARD? 6
2.3. WHAT IS THE ALTERNATIVE TO JAVA CARD? 6
2.4. WHAT DO I NEED TO WRITE JAVA CARD PROGRAMS? 6
2.5. WHAT IS THE JAVA CARD ARCHITECTURE? 6
2.6. HOW DOES JAVA CARD DIFFER FROM JAVA? 7
2.7. WHAT WILL BE ADDED TO THE NEXT REVISION? 7
2.8. WHO ARE THE MAIN JAVA CARD VENDORS? 8

3. JAVA CARD: APPLETS 9

3.1. WHAT IS A JAVA CARD APPLET? 9
3.2. CAN SEVERAL APPLETS BE PRESENT ON THE SAME CARD? 9
3.3. WHICH METHODS MUST I WRITE? 9
3.4. HOW DO I CHOOSE THE APPLET I'M SENDING COMMANDS TO? 9

4. JAVA CARD: CARD MANAGEMENT 10

4.1. HOW CAN I LOAD CODE ON A CARD? 10
4.2. JAVA CARD: CARD MANAGEMENT: OPEN PLATFORM 10
4.2.1. WHAT IS GLOBAL PLATFORM? 10
4.2.2. WHAT IS OPEN PLATFORM? WHAT IS VISA OPEN PLATFORM? 10
4.2.3. DO I REALLY NEED TO LEARN OPEN PLATFORM? 11
4.2.4. WHERE CAN I FIND THE OPEN PLATFORM SPECS? 11
4.2.5. WHAT IS THE CARD MANAGER? 11
4.2.6. WHAT IS A SECURITY DOMAIN? 11
4.2.7. WHAT IS A KEY SET? 11
4.2.8. HOW DO I LOAD AN APPLET USING OPEN PLATFORM? 11
4.2.9. WHAT IS SECURE MESSAGING? 12
4.2.10. HOW DO I OPEN/CLOSE A SECURE CHANNEL? 12
4.2.11. WHAT IS THE GLOBAL PIN? 12
4.2.12. HOW DO I CHANGE/UNBLOCK THE GLOBAL PIN? 12
4.3. JAVA CARD: CARD MANAGEMENT: OPEN PLATFORM 2.0.1 API 13
4.3.1. CAN I CHANGE THE VALUE OF THE GLOBAL PIN? 13
4.3.2. CAN I GET THE GLOBAL PIN STATUS (VERIFIED, NOT VERIFIED, BLOCKED)? 13
4.3.3. IS THE STATUS OF THE GLOBAL PIN LOCAL OR GLOBAL? 13

 

4.3.4. CAN I UNBLOCK THE GLOBAL PIN WITHOUT CHANGING IT? 13
4.3.5. HOW DO I OPEN/CLOSE A SECURE CHANNEL 13
4.3.6. HOW DO I DECRYPT/VERIFY INCOMING DATA ON A SECURE CHANNEL? 13

**5. JAVA CARD: CODE OPTIMIZATION 14**

**5.1. WHY SHOULD I OPTIMIZE THE APPLET CODE? 14**
**5.2. HOW CAN I REDUCE CODE SIZE? 14**
**5.3. HOW CAN I IMPROVE PERFORMANCE? 14**
**5.4. HOW CAN I REDUCE MEMORY CONSUMPTION? 15**

**6. JAVA CARD: COMMUNICATION 16**

**6.1. HOW DOES THE CARD TALK TO THE OUTSIDE WORLD? 16**
**6.2. WHAT IS AN APDU? 16**
**6.3. WHAT IS A STATUS WORD? 16**
**6.4. WHY CAN'T I JUST CALL 'THROW' TO RETURN THE STATUS WORD? 16**
**6.5. CAN I STILL RETURN DATA IF THE STATUS WORD IS NOT 0X9000? 17**
**6.6. IS RMI AVAILABLE? 17**
**6.7. CAN A MIDLET TALK TO A JAVA CARD APPLET? 17**

**7. JAVA CARD: CRYPTOGRAPHY 18**

**7.1. WHAT IS CRYPTOGRAPHY AND WHY DO I NEED IT? 18**
**7.2. WHERE CAN I LEARN MORE ABOUT CRYPTOGRAPHY? 18**
**7.3. WHAT CRYPTOGRAPHIC FEATURES DOES JAVA CARD SUPPORT? 18**

**8. JAVA CARD: DEVELOPMENT TOOLS 19**

**8.1. WHAT IS THE JAVA CARD DEVELOPMENT CYCLE? 19**
**8.2. WHAT IS A CAP FILE? 19**
**8.3. WHAT IS AN EXPORT FILE? 19**
**8.4. WHAT IS CONVERSION? 19**
**8.5. WHAT IS VERIFICATION? 20**
**8.6. HOW DO I DEBUG AN APPLET? 20**
**8.7. WHAT DOES THE SUN SDK CONTAIN? 20**
**8.8. WHERE CAN I FIND THE SUN SDK? 21**
**8.9. ARE THERE ANY JAVA CARD TOOL VENDORS? 21**
**8.10. HOW DOES WORKING WITH A SIMULATOR DIFFER FROM WORKING WITH A CARD? 21**

**9. JAVA CARD: DOCUMENTATION 22**

**9.1. WHAT KIND OF DOCUMENTATION IS THERE? 22**

 

**9.2. WHERE CAN I FIND THE JAVA CARD SPECS? 22**
**9.3. ARE THERE ANY BOOKS ON JAVA CARD? 22**
**9.4. WHAT DO I NEED BESIDES JAVA CARD DOCUMENTATION? 22**

**10. JAVA CARD: SECURITY 24**

**10.1. WHAT ARE THE SECURITY FEATURES OF JAVA CARD? 24**
**10.2. HOW SAFE IS JAVA CARD? 24**
**10.3. HOW DOES JAVA CARD PROTECT ITSELF AGAINST SPA, DPA, ETC? 24**
**10.4. DOES JAVA CARD SUPPORT BIOMETRICS? 24**
**10.5. JAVA CARD: SECURITY: OBJECT SHARING 25**
10.5.1. WHAT IS THE FIREWALL? 25
10.5.2. CAN TWO APPLETS SHARE DATA? 25
10.5.3. CAN ANY DATA BE SHARED? 25
10.5.4. HOW SECURE IS OBJECT SHARING? 25
**10.6. JAVA CARD: SECURITY: TRANSIENT DATA 26**
10.6.1. WHAT IS TRANSIENT DATA? 26
10.6.2. CAN ANY OBJECT BE TRANSIENT? 26
10.6.3. WHEN DO I NEED TO USE TRANSIENTS? 26
10.6.4. HOW MUCH TRANSIENT DATA CAN I ALLOCATE? 26
10.6.5. MAY TRANSIENT DATA BE SHARED? 26

**11. JAVA CARD: SMART CARDS 27**

**11.1. WHAT IS A SMART CARD? 27**
**11.2. WHAT ARE THE MAIN SMART CARD APPLICATIONS? 27**
**11.3. WHAT IS THE LIFE CYCLE OF A SMART CARD? 27**
**11.4. PLEASE EXPLAIN "MASK", "HARD MASK", "SOFT MASK" 28**
**11.5. ARE THERE ANY JAVA CARD PROCESSORS? 28**

**JAVA CARD: TRANSACTIONS 29**

**11.6. WHAT IS A TRANSACTION? 29**
**11.7. WHEN DO I NEED TO USE TRANSACTIONS? 29**
**11.8. HOW LARGE CAN A TRANSACTION BE? 29**
**11.9. ARE TRANSIENT WRITES TRANSACTIONED? 29**
**11.10. ARE NESTED TRANSACTIONS SUPPORTED? 30**
**11.11. WHAT IS THE SCOPE OF A TRANSACTION? 30**

 

# 1. Introduction

## *1.1 Scope*

Java Card is a software standard dealing with the execution of Java code on processor cards (a.k.a. smart cards). This FAQ deals with issues raised by the Java Card standard: specifications, API, tools, etc. It also covers related topics such as card security, cryptography and content management with Open Platform.

The Java Card FAQ is available online at http://www.jguru.com.

## *1.2 Maintainer*

The maintainer of the Java Card FAQ is Julien SIMON (jsimon@netattitude.fr). He is the founder of Netattitude (http://www.netattitude.fr).

Based in Paris, France, Netattitude is a software engineering company focusing on Java for the embedded space: digital TV, mobile computing, smart cards, etc.

Java Card

## *1.1. What is Java Card?*

Java Card is a software standard enabling the execution of Java code on smart cards. Java Card supports multiple platform-independent applications, as well as post-issuance loading.

The Java Card Forum is in charge of the Java Card standard (http://www.javacardforum.org). The current revision is 2.1.1 and was released on May 2000.

## *1.2. Why Java Card?*

The purpose of Java Card is two-fold:

- Simplify smart card application development: until recently, pretty much everything was coded in C and assembly, using proprietary card operating systems. Using a simple high-level language like Java on top of a standard API is a major improvement (at least we think it is!).
- Try to bring WORA (Write Once, Run Anywhere) to smart cards: since Java Card applications are hardware-independent, they may be loaded in binary form on any smart card that contains a Java Card platform.

## *1.3. What is the alternative to Java Card?*

There are three other options:

- **Proprietary card operating systems**.
- **MULTOS** (http://www.multos.com), a standard put together by Mondex International and now maintained by the MAOSCO Consortium (Europay, Mastercard, American Express, etc). Applications are written in a number of languages and compiled into MEL bytecode (Multos Elementary Language).
- **Windows for Smart Cards** (http://www.microsoft.com). Applications are written in Visual Basic and compiled.

## *1.4. What do I need to write Java Card programs?*

First of all, you don't need to be a smart card expert: that's the whole point of Java Card. With some experience of the Java language, you should be able to write Java Card applications pretty quickly. Here's what you'll need:

- A Java Card toolkit (see *Java Card:Tools*).
- The Java Card specifications (see *Java Card:Documentation*).
- JDK 1.3 or higher.
- A text editor.

As usual, a Java decompiler (like Jad - http://www.geocities.com/kpdus/jad.html) may also come in handy.

You don't need any hardware, like a card reader and actual cards. Of course, if you want to work on the real thing, you may purchase these from a number of vendors.

## *1.5. What is the Java Card architecture?*

 

The on-card components of the Java Card architecture are:

- The Java Card Virtual Machine, which interprets the application bytecode.
- The Java Card Runtime Environment, which interacts with the card reader and dispatches commands to Java Card applications.
- The Java Card Application Programming Interface, which export a number of services to Java Card applications. It is structured in four packages: *java.lang*, *javacard.framework*, *javacard.security* and *javacardx.crypto*.

There are also a number of off-card components, used to build and load applications on a card. See *Java Card:Tools* for more information.

Note that the Java Card standard doesn't cover card/application management, i.e. how applications are loaded, deleted, etc. See *Java Card:Card Management* for more information.

A Java Card platform usually sits on top of a minimal card operating system, which exports services for memory allocation, off-card communication, cryptographic operations, etc. The features and the API of this operating system are not part of the Java Card standard.

## *1.6. How does Java Card differ from Java?*

Massively. Both platforms are only similar in the sense that they use the Java language and run bytecode, but that's about it. Java Card is not a Java subset like J2ME. Java Card should rather be seen as a Java implementation targeted at smart cards:

- The Java Card API is totally specific, except for a couple of downsized java.lang classes.
- The Java Card Virtual Machine doesn't support any of these features: dynamic class loading, bytecode verification, Security Manager, threads, garbage collection, JNI, etc. It's not even able to process Java class files.

Still, if you know Java, you should pretty quickly feel comfortable with Java Card!

## *1.7. What will be added to the next revision?*

The next revision should be Java Card 2.2. The Java Card Forum doesn't communicate a lot (quite an understatement), so one can only guess what the schedule is. Early 2002 is as good a guesstimate as any.

A number of topics are being considered for inclusion in Java Card 2.2, but we'll never know until the standard is out. Here's a tentative non-exhaustive list:

- Java Card RMI (Remote Method Invocation), which is a subset of Java RMI for terminal-application communication.
- Support for new cryptographic algorithms like AES (Advanced Encryption Standard) and Elliptic Curves.
- Garbage collection.
- Transient objects.
- Object integrity and encryption.
- Threads.

 

## *1.8. Who are the main Java Card vendors?*

- Gemplus (http://www.gemplus.com)
- Giesecke & Devrient (http://www.gdm.de)
- Oberthur Card Systems (http://www.oberthurcs.com).
- Orga (http://www.orga.com)
- Schlumberger (http://www.slb.com)

 

# 2. Java Card: Applets

## *2.1. What is a Java Card applet?*

A Java Card application running on a smart card is called Java Card applet. Although they have the same name, the only thing a Java Card applet and a Java applet have in common is that both are objects. In other words, a Java Card applet doesn't have a *paint*() method…

An arbitrary number of classes and interfaces may compose an applet, as long as one of the classes extends *javacard.framework.Applet*. These classes and interfaces need to be stored in a package, which the converter transforms into a unique CAP file. The converter will also assign an AID to the applet. The CAP file may then be loaded and the applet may be instantiated. The applet may then be selected using its AID and sent commands to.

## *2.2. Can several applets be present on the same card?*

Yes. Java Card is a multi-application environment. Some applets may have been masked; some others may have been loaded post-issuance. However, only one applet is running at any given time.

## *2.3. Which methods must I write?*

Since an applet extends *javacard.framework.Applet*, it does inherit some methods, like *select*() and *deselect*(). The only methods which must be defined in your applets are *install*() and *process*().

If your applet implements object sharing, it must also implement *getShareableInterfaceObject*(). See *Java Card:Security:Object Sharing* for more information.

## *2.4. How do I choose the applet I'm sending commands to?*

In the normal case, commands are sent to the current applet, i.e. the last one that has been selected using the SELECT command. However, if the platform supports logical channels, multiple applets may be selected at the same time: dispatching is based on the value of the class byte in the APDU header.

 

# 3. Java Card: Card Management

## 3.1. *How can I load code on a card?*

Once you have converted the applet, you need to load the CAP file on the card. This means chopping the CAP file and send it as an APDU sequence to the on-card installer. Remember: the maximum amount of data than an APDU may hold is 255 bytes, so chances are your CAP file won't fit!

The nature of the APDU commands depends on the installer. The Java Card Runtime Environment Specification says very little about installation. What this means to you is that the installer that is part of your JCRE certainly uses proprietary commands.

This is a problem when you need to load an application on cards issued by different vendors. To alleviate this, most if not all Java Card implementations support the Open Platform standard.

## *3.2. Java Card: Card Management: Open Platform*

### 3.2.1. What is Global Platform?

GlobalPlatform is a cross-industry membership organization created to advance standards for smart card growth. It combines the interests of smart card issuers, vendors, industry groups, public entities and technology companies to define requirements and technology standards for multiple application smart cards. It is fully independent and democratic with priorities established by a Board of Directors. *(taken from the Global Platform FAQ - http://www.globalplatform.org)*

### 3.2.2. What is Open Platform? What is Visa Open Platform?

The Open Platform represents a set of cross-industry technical specifications, which can be used to develop secure, and flexible smart card systems. It includes both card and terminal specifications, as well as development tools. Together, these components define an easy-to-use smart card platform upon which standardized applications can be added, such as credit, debit, electronic purse, and loyalty points, as well as applications that support opportunities in a variety of industry segments. The Open Platform works across different cards and operating systems. It enables smart card issuers to choose between operating systems and application developers while providing a core security and card management technology. Originally defined by Visa, the Open Platform has evolved into cross-industry specification for complete multiple application smart card management. The Open Platform is now owned, managed and developed by GlobalPlatform. *(taken from Global Platform FAQ - http://www.globalplatform.org)*

The most recent version is 2.1 (June 4th, 2001). However the most widely deployed version is 2.0.1 (April 7th, 2000)

 

### 3.2.3. Do I really need to learn Open Platform?

Yes. Most vendors and application providers have Open Platform compliant smart cards. If you are a developer or a tester, the odds of working on OP compliant applet are pretty high.

### 3.2.4. Where can I find the Open Platform specs?

The Open Platform Card Specification for versions 2.0.1 and 2.1 may be downloaded from http://www.globalplatform.org.

### 3.2.5. What is the Card Manager?

The Card Manager is the central entity in the Open Platform Standard. It's in charge of processing all Open Platform commands for installation, life cycle management, security, etc.

The Card Manager is usually a Java Card applet, which means that it has its own AID and may be selected. Most of the time, the Card Manager is the default applet, so it's implicitly selected when the card is powered up or reset.

### 3.2.6. What is a Security Domain?

A Security Domain is a non-selectable applet used to store another applet's cryptographic keyset(s). Each Java Card applet must be associated to a Security Domain (either its own or the default one). This association is performed at installation time and cannot be broken.

If you want your applet to use its own Security Domain, you have to implement the latter yourself, i.e. implement the *ProviderSecurityDomain* interface.

### 3.2.7. What is a Key Set?

A key set is composed of 3 keys:

- **Authentication Key**: the authentication key is used to generate an encryption session key. The encryption session key will be used to encrypt an APDU data field sent over a secure channel using data encryption.
- **MAC Key**: the MAC key is used to generate a MAC session key. The MAC session key will be used to compute the MAC on an APDU sent over a secure channel using data integrity.
- **Key Encryption Key** (KEK): the KEK is used to encrypt an APDU data field. This provides an additional level of encryption when loading sensitive data on the card (like cryptographic keys). The KEK can be used to encrypt data inside or outside a secure channel, whereas the authentication and MAC keys are used within the context of a secure channel.

### 3.2.8. How do I load an applet using Open Platform?

First, you need to send the CAP file to the card. Whatever toolkit you use, it should include a tool that builds a load script from the CAP file. In other words, this tool chops

 

the CAP file and writes a script composed of an INSTALL LOAD command, followed by as many LOAD commands as needed to send the CAP file.

Once the CAP file has been loaded, you have to instantiate the applet using the INSTALL INSTALL command. This will trigger a call to the applet's *install*() methods, which must the applet constructor and register the applet with the JCRE.

The final step is to make the applet selectable using the INSTALL MAKE SELECTABLE command. You don't have to run this command immediately, but only when you actually want the applet to be selectable.

Note that these last two steps may be condensed into one, in which case the applet is instantiated and immediately made selectable.

### 3.2.9. What is Secure Messaging?

Secure Messaging is maybe the most widely used feature in Open Platform. It enables a reader to open a secure communication link with an applet, using one of the key sets of the applet. This link is called secure channel and may be opened in two mores, usually called SM-MAC and SM-ENC.

SM-MAC guarantees data integrity by appending a MAC (Message Authentication Code) to the APDU data. The Card Manager performs integrity checking.

SM-ENC adds data encryption, so that it may not be eavesdropped. The Card Manager performs data decryption.

### 3.2.10. How do I open/close a Secure Channel?

Opening a Secure Channel is performed using a sequence of two APDU commands: INITIALIZE UPDATE and EXTERNAL AUTHENTICATE. These commands are defined by the Open Platform specs.

### 3.2.11. What is the Global PIN?

The Global PIN is a PIN that may be checked by all applets on a card, using *OPSystem.verifyPin*(). Its value is usually set at personalization time.

### 3.2.12. How do I change/unblock the Global PIN?

There is no APDU command to change/unblock the Global PIN in the Global Platform specification. However, the Visa Card Implementation Requirements define a command to do so.

 

## *3.3. Java Card: Card Management: Open Platform 2.0.1 API*

### 3.3.1. Can I change the value of the Global PIN?

An applet may only change the value of the Global PIN if it was granted the PIN CHANGE privilege. Applet privilege is specified at installation time: it's sent as a byte in the INSTALL command.

### 3.3.2. Can I get the Global PIN status (verified, not verified, blocked)?

No. There is no method equivalent to *PIN.isValidated()* in the OP API. The applet needs to save the GP status locally. Using transient data to do so is a good idea.

### 3.3.3. Is the status of the Global PIN local or global?

The status of the Global PIN is local to each applet. However, the Global PIN value and try counter are shared by all applets.

### 3.3.4. Can I unblock the Global PIN without changing it?

No, the only available method is *OPSystem.setPin*() which changes the value of the Global PIN and unblocks it.

### 3.3.5. How do I open/close a Secure Channel

To implement INITIALIZE UPDATE and EXTERNAL AUTHENTICATE, you respectively have to use:

- *ProviderSecurityDomain.openSecureChannel*()
- *ProviderSecurityDomain.verifyExternalAuthenticate*()

Closing a secure channel may be performed explicitly with *ProviderSecurityDomain.closeSecureChannel*(). Other events will implicitly close a secure channel: card reset, wrong MAC, etc.

### 3.3.6. How do I decrypt/verify incoming data on a Secure Channel?

You just have to pass the incoming APDU to *ProviderSecurityDomain.unwrap*(). If the call returns, this means that the APDU has been successfully decrypted/verified. If not, the Card Manager throws an exception and closes the Secure Channel.

# 4. Java Card: Code Optimization

## *4.1. Why should I optimize the applet code?*

There are three main reasons why you have to optimize your code:

- Resources are extremely constrained. EEPROM size typically is 32Kb, and is used to hold post-issuance applets as well as application data. RAM size is usually about 2Kb, and must accommodate for application stack, transient data and the APDU buffer. You just can't afford any waste.
- Processing power is very limited: the chip usually holds an 8-bit CPU running at 5MHz. Also, Java Card applets are much slower than native code, so you don't want to perform unnecessary operations.
- Even though your applet runs fine on your platform, you still want to save as much resources as possible, so that it will also run on more limited platforms (less stack, smaller transaction buffer, etc). WORA, remember?

## *4.2. How can I reduce code size?*

Code size is particularly important for post-issuance applets: they are loaded in EEPROM, so the bigger they get, the less application data you can store.

Here are a number of tips that help reduce code size:

- Keep the number of classes/interfaces to a minimum. Class overhead is over 200 bytes, so fancy OO designs with plenty of patterns and interfaces are not appropriate.
- Keep the number of methods to a minimum. Increasing member visibility eliminates the need for the usual get/set methods. Not very nice, but efficient.
- Try to use no more than 3 parameters for virtual methods and 4 for static methods. This way, the compiler will use a number or bytecode shortcuts, which help reduce code size: *aload_x* (1 byte) instead of *aload x* (2 bytes), etc.
- Use the switch-case control structure with care. It generates very verbose bytecode.
- Reuse local variables instead of declaring new ones.
- Hunt down unused but initialized local variables.

## *4.3. How can I improve performance?*

Here are a number of tips that help speed up execution:

- Use *arrayFillNonAtomic*/*arrayCopy*/*arrayCopyNonAtomic* to initialize/modify arrays. These are native methods, which are way faster than any clever Java code you could come up with.
- In general, whenever the platform provides native code to perform an operation, use it!
- Avoid EEPROM writes. Writing in EEPROM is about 1,000 times slower than writing in RAM, so the less you do it, the better. You'll save tens of milliseconds, if not hundreds. Believe us.
- Use transient arrays to store session data and temporary results. Transient arrays are stored in RAM, so you'll avoid costly EEPROM writes.
- Use the APDU buffer for cryptographic operations. Ditto: the APDU buffer is stored in RAM.

 

- Avoid deep class trees: the deeper the tree, the slower the search for virtual methods. This can cost you several milliseconds on each method invocation…
- Save *array.length* in a local variable before using it in a loop. If the loop is long enough, this will make a difference.
- Do not use exceptions for flow control, only for error handling. Exception processing is very slow.

## 4.4. *How can I reduce memory consumption?*

Here are a number of tips that help conserve EEPROM:

- When possible, allocate all applet resources in the constructor. This is interesting for two reasons. First, if there's not enough memory to install your applet, you'll know immediately. This is much better than to face a memory allocation at runtime, when the card has already been delivered… Second, allocation memory means writing to the EEPROM, which is slow.
- The Java Card standard doesn't require garbage collection. So unless you are willing to waste memory, you have to keep track your old objects and reuse them. Remember: the Java Card Virtual Machine runs "forever", so if an object becomes unreachable, its memory is gone "forever" (or at least until you delete the applet).
- Even if your platform provides proprietary garbage collection, you'd better reuse your objects. Allocating new objects is slow, and so is garbage collection. You get the point.
- Use a big buffer instead of several small buffers. Depending on how your platform allocates memory, 4 byte arrays of 32 bytes may eat more EEPROM than a unique byte array of 128 bytes.
- To avoid stack overflows: avoid deep class tress, reuse local variables and keep the number of parameters to the strict minimum.

Bottom line: it pays to recycle!

 

# 5. Java Card: Communication

## *5.1. How does the card talk to the outside world?*

A card needs to be inserted in a card reader (also called card acceptance device, or CAD). If the card is a contact less card, it just needs to be close enough from the reader for a fixed period of time.

The card reader always initiates communication. It provides power and clock to the card. First, it will usually select a card application using a special command called SELECT. Then, it sends a command to the application, which processes it and answers. The reader sends the next command, and so on. Simple, huh?

Commands and answers are structured as APDUs (Application Protocol Data Unit). These APDUs are sent back and forth using either one of two protocols, oddly named T=0 and T=1. Whatever protocol is used is transparent to Java Card applications, so you can safely ignore this for now. If you really need more information, refer to the ISO 7816-3 standard.

## *5.2. What is an APDU?*

The APDU (Application Protocol Data Unit) is the communication unit between a reader and a card. The structure of an APDU is defined by the ISO 7816 standards.

There are two categories of APDUs: command APDUs and response APDUs. As the name implies, the former is sent by the reader to the card: it contains a mandatory 5-byte header and from 0 to up to 255 bytes of data. The latter is sent by the card to the reader: it contains a mandatory 2-byte status word and from 0 to up to 256 bytes of data.

## *5.3. What is a status word?*

The status word is a 2-byte value, returned to the reader after an application has processed an APDU. Data is optional in a response APDU, but the status word is mandatory.

If the *process*() function runs to completion, status word 0x9000 is implicitly returned (meaning everything is OK). If the application needs to return another value (probably to signal an error condition), it can do so by invoking the *throwIt(short value)* method in the *ISOException* class.

*javacard.framework.ISO7816* defines a number of frequently used values, but you're free to use anything you'd like, provided that it's not already used by the platform itself.

## *5.4. Why can't I just call 'throw' to return the status word?*

In standard Java, you'd do something like "*throw new MyException*()". The problem is that Java Card doesn't support garbage collection. Hence, it's highly undesirable to create many short-lived objects, whose memory will NOT be reclaimed. Here, "many" really means one for each APDU!

 

Instead, you must call *ISOException.throwIt*(*myValue*). This class extends the Exception class and implements the well-known Singleton pattern, so there's only one instance of the *ISOException*, which is reused. This way, applets can throw and catch exceptions, but still conserve memory.

## *5.5. Can I still return data if the status word is not 0x9000?*

Yes. Data is only transmitted at the end of the *process*() function, so as long as you call *apdu.setOutgoing*() before throwing the *ISOException*, there's no problem.

## *5.6. Is RMI available?*

Not yet. Right now, you have to use ISO7816-style communication. However, a subset of Java RMI is strongly considered for Java Card 2.2.

## *5.7. Can a midlet talk to a Java Card applet?*

No. Currently, the MID profile (http://java.sun.com/products/midp/) doesn't provide any API that would let a midlet running on a J2ME/MIDP phone talk to a Java Card applet running on the SIM card. It's really a shame, because since this would certainly lead to many interesting applications.

However, several Java Card vendors have expressed deep interest in this matter (notably at Java One 2001) and hopefully, this need will be addressed through the Java Community Process (http://www.jcp.org/). Other parties seem to disagree, so don't hold your breath…

 

# 6. Java Card: Cryptography

## *6.1. What is cryptography and why do I need it?*

Cryptography is mostly known as the art of encryption, i.e. protecting confidential data by transforming it into something unintelligible. However, cryptography is also the basis for authentication, digital signatures, etc. All these techniques are crucial to current information systems and they are vital to smart cards. You couldn't write real-life smart card applications without cryptography

## *6.2. Where can I learn more about cryptography?*

Cryptography is a fascinating field. Here are a few pointers to get you started:

- **The Cryptography FAQ** at RSA Labs: http://www.rsa.com/rsalabs/faq.
- **Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition**, by Bruce Schneier, Wiley, John & Sons, Incorporated; 10/1995. This is the bible. There's also a French-language translation: "Cryptographie Appliquée".
- **Cryptography Research, Inc**.: http://www.cryptography.com. Lots of cryptographic resources and state-of-the art information on attacks.

## *6.3. What cryptographic features does Java Card support?*

Java Card 2.1.1 offers the following cryptographic features:

- *Javacardx.crypto.Cipher* provides encryption/decryption, using either secret-key algorithms (DES, Triple DES) or public-key algorithms (RSA).
- *Javacard.security.Signature* provides signature/verification using either secret-key algorithms (DES, Triple DES) or public-key algorithms (RSA, DSA).
- *Javacard.security.MessageDigest* provides message digests, using one-way hash functions (MD5, SHA-1, RIPEMD160).

Please refer to the Java Card specification for more information on supported key lengths, modes of operations and padding.

Note that Java Card 2.1.1 supports neither AES (Advanced Encryption Standard) nor Elliptic Curves. However, they probably will be part of Java Card 2.2.

 

# 7. Java Card: Development Tools

## *7.1. What is the Java Card development cycle?*

The development cycle of a Java Card applet goes like this:

- Write the Java source;
- Compile it using your favorite Java compiler;
- Convert the resulting class files into a unique CAP file;
- Verify that the CAP file is valid;
- Load the CAP file either on an actual card or in a software simulator;
- Instantiate the applet;
- Select the applet and send commands to it.

## *7.2. What is a CAP file?*

The CAP file format is defined by the *Java Card 2.1.1 Virtual Machine Specification*.

A CAP file is the loading unit in Java Card. Using a host tool called converter, a CAP file must be built for each package that will be loaded on the card: it will contain all classes and interfaces belonging to the same package.

Although CAP means "converted applet", a CAP file doesn't necessarily contain an applet. If the corresponding package is a simple library (i.e. it only defines an API), there won't be any applet inside the CAP file…

## *7.3. What is an export file?*

The EXP file format is defined by the *Java Card 2.1.1 Virtual Machine Specification*.

An EXP file provides information on the public API of a package, i.e. the signature of its public methods. It is built when the package is converted and is mainly used for linking purposes.

## *7.4. What is conversion?*

Conversion is the operation through which the class files of a package are turned into a CAP file. A host tool performs this operation: the converter.

The main reasons for conversion are:

- The Java Card Virtual Machine doesn't support dynamic class loading. Thus, one must make sure that all classes are "pre-loaded" on the card before the application runs.
- Card resources are extremely constrained, so costly operations like linking are better left performed on the host.
- Some Java features are either optional or missing in Java Card (integer support, static initializers, etc); one must make sure that class files conform to the actual Java Card platform they are meant for.
- An AID must be assigned to the package and the application.

 

Most Java Card vendors have implemented their own converter, mainly because Sun's converter wasn't available/reliable for a while. In theory, since the CAP file format is standardized, you could use any converter and then load the CAP file on any card. In practice, we have found at least one combination that doesn't work at all and crashes the card… So, if you plan on using a different converter than the one provided by your vendor, run a few tests to make sure they really are equivalent.

## *7.5. What is verification?*

Verification is the operation through which a CAP file is checked against the CAP specification and the export files of the packages it references. A host tool performs this operation: the verifier.

Java Card support post-issuance loading, but since the Java Card Virtual Machine doesn't support verification, this last step must be performed before a CAP file is loaded on the card. The purpose is two-fold. First, make sure that a CAP file is valid (structure, bytecode, etc). Second, make sure that its methods interact correctly with other packages (number and type of parameters, etc). Please note that verification doesn't guarantee the origin of the CAP file.

Most Java Card vendors have implemented their own verifier, because Sun's verifier wasn't widely available until April 2001.

## *7.6. How do I debug an applet?*

Right now, you most likely can't debug a Java Card applet at source level, because the Java Card Virtual Machine you're working on doesn't support the Java Platform Debug Architecture (JPDA - http://java.sun.com/j2se/1.3/docs/guide/jpda/) and thus cannot be controlled by your favorite debugger. It looks like the Java Card community has acknowledged this problem and is taking steps to improve the situation (hopefully).

In the meantime, what can you do? Here are a few tips:

- Code more carefully: Java Card is an embedded environment and you really have to be much more cautious, so stay focused!
- Try to run most of your code in Java land, not in Java Card land. This way, you can easily debug your application logic and then run it safely on Java Card.
- When running on the card, use *ISOException.throwIt*((short)((short)0x9000+myValue)) to inspect a variable.
- If your simulator provides access to the bytecode executed by the Virtual Machine, you can really see a lot of what's happening. Go to the *Java Card 2.1.1 Virtual Machine Specification* for the meaning of each bytecode instruction. It's just like debugging assembly language!
- If you think you've stumbled upon a platform bug, there's nothing you can do unless you have an emulator, mask sources, etc. Go bug (pun not intended) whoever has that.

## *7.7. What does the Sun SDK contain?*

The Java Card 2.1.2 SDK, released in April 2001, contains:

 

- Two Java Card simulators: JCWDE (Java Card Workstation Development Environment), which is written in Java, and CREF, which is a native executable. The main difference is that the former doesn't support dynamic application loading, while the latter does.
- Class files and export files for the Java Card API.
- Development tools: a converter, an off-card verifier, tools to dump the contents of CAP and EXP files, tools to send commands to the simulator, etc.
- Sample applets.
- Tools documentation.

The current release doesn't provide any cryptographic features. Although most of the classes are present, you won't be able to build keys, which makes it pretty difficult to perform encryption and signatures. The root cause seems to be export restrictions enforced by the U.S. Government.

## *7.8. Where can I find the Sun SDK?*

The Java Card 2.1.2 SDK may be downloaded from http://java.sun.com/products/javacard.

## *7.9. Are there any Java Card tool vendors?*

As far as we know, the only company working on a Java Card IDE is Metrowerks (http://www.metrowerks.com). Code Warrior for Java Card is currently under development and supports source-level debug. We are currently beta-testing it and it looks promising. Metrowerks aim for an October 2001 release.

There are also a number of vendor-specific developments toolkits targeted at GSM applications. Check out their websites for more information.

## *7.10. How does working with a simulator differ from working with a card?*

 

# 8. Java Card: Documentation

## *8.1. What kind of documentation is there?*

The Java Card specification is composed of three documents:

- *The Java Card 2.1.1 Virtual Machine Specification* focuses on the features of the Java Card Virtual Machine, the CAP file format and the Java Card bytecode.
- *The Java Card 2.1.1 Runtime Environment Specification* focuses on the features of the Java Card environment (applets, transactions, security, etc).
- *The Java Card 2.1.1 Application Programming Interface* is in javadoc format and details the Java Card API.

If you are new to smart cards and thus to Java Card, we do not recommend that you dive immediately into these specs. Our feeling is that although they are the reference documentation, they do no provide an easy learning path. You would probably be better off reading a Java Card book first, and then move on the specs.

## *8.2. Where can I find the Java Card specs?*

The Java Card 2.1.1 specifications may be downloaded in PDF format from http://java.sun.com/products/javacard.

## *8.3. Are there any books on Java Card?*

Yes. Here are the ones we are aware of. They may be purchased from all good online bookstores, like Fatbrain (http://www.fatbrain.com) or Amazon (http://www.amazon.com).

***Java Card Technology for Smart Cards: Architecture and Programmer's Guide* by Zhiqun Chen; published by Addison Wesley Longman, Inc., 06/2000.**
This is our preferred starting point for beginners. This book covers smart card basics and provides a gentle introduction to Java Card and its main features. It is quite programmer-oriented and provides many code samples, which will help you get started very quickly. However, this book is not thorough enough to serve as a reference book in the long run.

***Smart Card Application Development Using Java*, by Uwe Hansmann, Frank Seliger,Martin S. Nicklous; published by Springer-Verlag New York, Incorporated, 12/1999.**
This book provides a high-level of the Java Card environment: features, applications, the Open Card Framework, etc. Although this is a very serious book, which is definitely worth reading, we don't feel it is appropriate for beginners. It probably targets project leaders or managers who want to get a good grasp on Java Card.

## *8.4. What do I need besides Java Card documentation?*

Java Card is just part of a larger picture. If you plan on writing real-life Java Card development, you'll definitely need:

- The **Open Platform** specs. See *Java Card:Card Management:Open Platform* for pointers.

 

- A good book on **cryptography**. See *Java Card:Cryptography* for pointers.
- The **ISO7816** standards: the relevant ones are ISO7816-4, ISO7816-8 and ISO7816-9. These standards define APDU commands related to file management, cryptographic operations, etc. Many applets need to implement at least a couple of these, so these standards are mandatory reading. They are not freely available and must be purchased from ISO (http://www.iso.ch).

If you're writing applets for the GSM world, you'll need these two standards. They are not freely available and must be purchased from ETSI (http://www.etsi.org):

- **GSM11.11**: Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface.
- **GSM 11.14**: Digital cellular telecommunications system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface.

If you're writing applets for the banking world, you'll need the **EMV** standards. They may be freely downloaded from the EMVCo website (http://www.emvco.com).

If you are also interested in client-side software, you may want to look at the **Open Card** standard (http://www.opencard.org).

 

# 9. Java Card: Security

## *9.1. What are the security features of Java Card?*

The main security features of Java Card are:

- All the benefits of the Java language: data encapsulation, safe memory management, packages, etc.
- Applet isolation, thanks to the Java Card firewall.
- Transient data, which guarantees that sensitive session data is wiped out.
- A rich cryptography API for encryption, digital signatures and message digests. See *Java Card:Cryptography*.
- Secure communication with the card reader, if the card is Open Platform compliant. See *Java Card:Card Management:Open Platform*.

## *9.2. How safe is Java Card?*

This is a very tough question. The security of Java cards is evaluated using the Common Criteria (CC) methodology. A couple of platforms (Gemplus & Oberthur Card Systems) have passed security level EAL1+. No Java Card platform has passed the EAL4+ level yet: this level is the minimum safety level requested by banking applications. This is where Multos has a key advantage over Java Card, so you can bet that Java Card vendors are hard at work trying to reach it. Let's wait and see!

If you want to learn more on CC, go to:

- The CC web site: http://www.commoncriteria.org.
- The NIST page on CC for smart cards: http://csrc.nist.gov/cc/sc/sclist.htm.

Beyond that, one of the overall concerns seems to be the lack of on-card verification. Some argue that it's unnecessary if an Open Platform compliant Card Manager enforces secure applet loading. Some argue it is still necessary and claim that on-card verification is possible using Proof-Carrying Code (http://www.cs.berkeley.edu/~necula/pcc.html).

## *9.3. How does Java Card protect itself against SPA, DPA, etc?*

The Java Card specification doesn’t deal with smart card attacks. Card issuers have to make sure that their Java Card implementation is immune (or at least very resistant) to all well-known smart cards attacks like timing attacks, SPA, DPA or DFA.

## *9.4. Does Java Card support biometrics?*

No. As of version 2.1.1, Java Card doesn’t provide any biometrics API.

 

## *9.5. Java Card: Security: Object Sharing*

### 9.5.1. What is the firewall?

The firewall is a software feature of the Java Card platform, which isolates applets from each other. In other words, even if an applet succeeds in obtaining a reference to an object belonging to another applet, the platform guarantees that any attempt to use this reference will trigger an exception.

The firewall attaches a memory context to package present on the card: thus, the context of an applet is the context of the package in which the applet lives. When an object is accessed, the Java Card Virtual Machine compares the object context (i.e. the context of the applet that created it) to the context of the current applet. If they don't match, the VM throws a *SecurityException*.

### 9.5.2. Can two applets share data?

Yes. If two applets live in the same package, they are attached to the same memory context. Thus, they may exchange object references and use them directly. Two applets living in different packages may also share data, but in a much more controlled way: this is called object sharing and involves quite a bit of work!

### 9.5.3. Can any data be shared?

No. To be shareable, an object must be an instance of a class implementing an interface extending the Shareable interface. Got that? Well, this is the one-line version. Check out the Java Card documentation for more information and come back here when you have more questions!

### 9.5.4. How secure is object sharing?

The security of object sharing relies on AIDs. If an applet may be loaded with an arbitrary AID, then security is pretty much out the window. This is why the security mechanisms provided by Open Platform are so important. Beyond that, we still face a few issues:

- Once a shareable object reference has been granted to a client, it can't be revoked. This notably means that even if the server has released the object, it can't be garbage-collected if the client hasn't done so.
- There is no way to prevent a legitimate client from acting as a proxy for hostile clients (man-in-the-middle attack).
- If a shareable object implements several shareable interfaces, we can't restrict a client to just one of these interfaces. Thus, this kind of design should be avoided.

 

## *9.6. Java Card: Security: Transient Data*

### 9.6.1. What is transient data?

Transient data is applet data, which is cleared either when the applet or deselected or when the card is reset. Transient data is stored in a dedicated RAM area and is typically used to store session data (like Secure Messaging session keys). The transient API is defined in *javacard.framework.JCSystem*.

### 9.6.2. Can any object be transient?

No. Java Card 2.1.1 only support transient arrays of primitive types (boolean, byte, short) and of *Object* references. You can't just create a transient instance of the *MyObject* class.

Note than only the contents of a transient array is cleared. The reference on the array itself is stored in EEPROM, so there's no need to reallocate it at the beginning of each session.

### 9.6.3. When do I need to use transients?

Transients are relevant when you need to store data that mustn't survive either applet deselection or card reset. They're also good for frequently modified non-persistent data and temporary results, because they save costly EEPROM writes.

### 9.6.4. How much transient data can I allocate?

Not much. The value is platform-dependent, but it's usually around 220 bytes. Transient space is shared among all applets, so you need to be very conservative. Unfortunately, there's no API to get the amount of transient space that the platform supports.
As a consequence, try to stick to the CLEAR_ON_DESELECT type, because since this type of data is meaningless when the applet is deselected, the platform should be able to reuse the space for the current applet.

### 9.6.5. May transient data be shared?

No. Any attempt to share a transient array, whatever its type is, will trigger a *SecurityException*.

 

# 10. Java Card: Smart cards

## *10.1. What is a smart card?*

Cards come in two flavors: memory cards and processor cards. Memory cards have no CPU and thus have no processing power: they are just used to store data. Processor cards have a CPU, which enables them to run on-board applications: that's why they are called "smart cards".

Most cards need to be inserted into a card reader (also called card acceptance device, or CAD). However, some cards are contact less and simply need to be close enough from a reader for a given period of time.

For contact cards, the visible golden part is not the chip: these are the electrical contacts, through which the card is powered and can talk to the outside world. If you want to see the chip, you have to remove the contacts (THIS WILL DESTROY YOUR CARD!). The chip holds the following items:

- The CPU. It usually is an 8-bit processor, running at an internal clock speed of 5MHz.
- An optional cryptographic coprocessor, to speed up DES/RSA crypto operations.
- ROM (Read-Only Memory): it can't be altered and holds the runtime environment as well as default applications. Typical amount is 64Kb.
- EEPROM (Electrically Erasable Programmable Read-Only Memory): it's persistent and holds applications loaded after the card has been issued, as well as application data. Typical amount is 32Kb.
- RAM (Random Access Memory): it's volatile and holds application stack, as well as transient data. Typical amount is 2Kb.
- I/O lines, to enable the card to talk to the reader. This is typically a set of serial lines.

Future cards will use a 32-bit RISC processor, Flash Memory or FeRAM instead of EEPROM, and USB instead of serial lines.

## *10.2. What are the main smart card applications?*

Chances are you have currently at least one smart card in your pocket right now! The main fields where smart cards are used are banking, mobile telephony (GSM), pay-TV, e-commerce (e-wallet, B2B, etc) and security (corporate badge, etc).

If you need more information, just visit the websites of card issuers. They usually have pretty nice marketing slides…

## *10.3. What is the life cycle of a smart card?*

The **card issuer** (Schlumberger, Gemplus, Oberthur Card Systems, etc) writes the runtime environment and sends it to the chip manufacturer. The **chip manufacturer** (Philips, ST Micro, Infineon, etc) burns runtime environment into the ROM of the chips: this is called **masking**. Once the chips are ready, they are sent back to the card issuer, who glues a chip on each card and loads application data: this last step is called **pre-personalization**.

 

Once the cards have been pre-personalized, they are delivered to the **application provider** (Visa, American Express, etc). The application provider prints and embosses the cards. Then, they usually load their own applications (hence the term post-issuance loading) and their own application data: this step is called **personalization**.

Finally, the application provider sends the card to the **cardholder**, i.e. the end user, which will use the card until the card is destroyed, the service is terminated, etc.

## *10.4. Please explain "mask", "hard mask", "soft mask"*

In the smart card context, the mask is the software that is burned in ROM by the chip manufacturer. This will include the runtime environment and the default applications provided by the card issuer. Some people also use the terms “hard mask” and “ROM mask”.

Using a devilish mechanism, it’s possible to patch part of the hard mask with code stored in EEPROM: this code is called “soft mask”. Its purpose is usually to fix bugs in the hard mask: you just shipped the new mask to the manufacturer and an intern finds a critical bug (sounds familiar?). Nobody wants to throw 100,000 brand new cards away, so you’ll have to come up with a soft mask. This will save the day, but it’ll waste a few kilobytes of precious EEPROM…

## *10.5. Are there any Java Card processors?*

Yes. Several CPUs are able to Java Card bytecode directly. Here are the ones we know of:

- **SmartJ ST22** - ST Micro (http://www.st.com/).
- **Jsmart** – Nazomi (http://www.nazomi.com/).

 

# Java Card: Transactions

## *10.6. What is a transaction?*

A transaction is a set of modifications performed atomically, which means that either all modifications are performed or none are performed. This is particularly for smart cards, because the card reader powers them: when you unexpectedly remove the card from the reader (this is called "tearing"), it's possible that you're interrupting a critical operation that needed to run to completion. This could put the card in an irrecoverable state and make it unusable.

To prevent this, the Java Card platform offers a transaction mechanism, through which the developer may declare a set of modification as being part of a same transaction. If the transaction completes normally, then all writes are committed. If the transaction is interrupted (power loss, application error, etc), previous writes will be undone automatically when the card is next powered up. The transaction API is defined in *javacard.framework.JCSystem*.

## *10.7. When do I need to use transactions?*

You need to use transactions anytime you're modifying a set of data, which could end up in a weird state if the modifications don't run to completion.

For example, changing the value of a PIN needs to be transactioned: you wouldn't want to change only the first two bytes, would you?

Another example: when an account is credited, another is debited from the same amount. Crediting the first account and debiting the second account must be performed atomically.

A number of Java Card APIs use the transaction facility. The most notable is *javacard.framework.arrayCopy*().

You don't need to use transactions when you're modifying a single primitive value (primitive types, one cell of an array of primitive types). The Java Card platform guarantees that these writes are atomic. However, please note that a set of atomic writes is not itself atomic, hence the need for transactions!

## *10.8. How large can a transaction be?*

This is platform dependent. You may use *JCSystem.getMaxCommitCapacity*() to find out what this amount is. Because of overhead, the size of the transaction buffer is not fully available to applets, so chances are you'll get a *TransactionException* before you actually write this much data in your transaction…

## *10.9. Are transient writes transactioned?*

 

No. Transient data is lost whenever the card is powered-up, so it wouldn't make sense to include it in transactions.

## *10.10. Are nested transactions supported?*

No. Because of the limited resources of current smart cards, Java Card 2.1.1 only supports one transaction level. If you call *JCSystem.beginTransaction*() when a transaction is already is progress, the VM will throw a *TransactionException*.

Using *JCSystem.getTransactionDepth*(), you can find out if a transaction is currently opened.

## *10.11. What is the scope of a transaction?*

Only a successful call to *JCSystem.commitTransaction*() will commit a transaction.

If a transaction is not explicitly committed, it will abort at the end of the *process*() method. This means that a transaction cannot span across multiple APDUs. The current transaction will also abort if an exception is thrown, or if the transaction buffer is full.