La technologie javacard.

Quelques rappels sur la technologie Java.

La technologie java est organisée autour d'objets (java), matérialisés par des fichiers *ClassFile* (dont l'extension est *.class*) , obtenus après compilation d'un fichier source (.java) par le compilateur *javac*.

Le *code byte* java utilise des index pour référencer les objets, les méthodes (*methods*) et les variables (*fields*). Une classe est identifiée par un nom (*Fully Qualified Name*) dont la partie gauche désigne un *package* (paquetage, un chemin d'accès) et dont la partie droite représente le nom de la classe.

La table *constant_pool* établit une relation entre un index et une information (par exemple le nom d'une classe…). Lors du chargement et de l'exécution d'une classe, la machine virtuelle java (JVM) réalise une nouvelle table (*runtime constant_pool*) qui permet de lier la classe avec l'environnement courant.

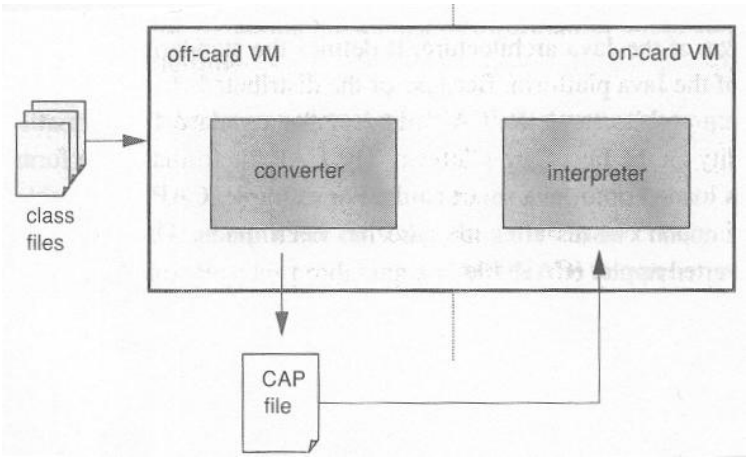
ClassFile {
u4 magic; // 0xCAFEBABE ;
u2 minor_version ; // plus petite version supportée
u2 major_version ; // plus grande version supportée
u2 constant_pool_count ; // nombre d'entrée de la table + 1
cp_info constant_pool[constant_pool_count-1] ; // table des constantes *cp_info*
u2 access_flag; // privilèges d'accès et propriétés de cette classe
u2 this_class ; // index de classe dans la table des constantes
u2 super_class ; // index de la super classe dans la table des constantes.
u2 interfaces_count ; // le nombre d'interfaces
u2 interfaces[interfaces_count] ;// tableau d'index dans la table des constantes.
u2 fields_count ; // nombres de variables
field_info info[fileds_count] ; // Tableau de descripteurs *field_info* des variables
u2_method_count ; // nombre de méthodes
method_info methods[methodes_count] ; // tableau de descripteurs *method_info*
u2 attributes_count ;//nombre des attributs (valeurs variables, code des méthodes …) de la classe.
attributes_info attributes[attributes_count] ; // tableau des valeurs (*attributes_info*) des attributs
}

Dans l'univers java les classes sont stockées dans des répertoires particuliers (identifiés par la variable d'environnement *classpath*) ou dans des serveurs WEB. L'adaptation de la technologie java aux cartes à puce implique en particulier une adaptation des règles de localisation des classes à cet environnement particulier.

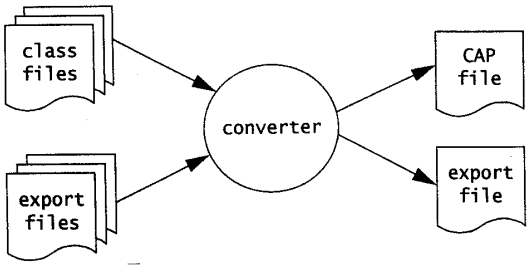
La norme JavaCard 2.x

Le langage *javacard* est un sous ensemble du langage java (paquetage *Java.lang)*. Il supporte les éléments suivants,

Eléments supportés.	Principaux éléments non supportés
Les types primitifs boolean, byte short. Le type primitif int est optionnel	Types primitifs long, double, float, char
Tableau à une dimension	Tableau à plusieurs dimensions
Paquetages, classes, interfaces exceptions. Héritages, objets virtuels, surcharge, création d'objets (new).	Les objets String. Chargement dynamique de classe (new lors du *runtime*)

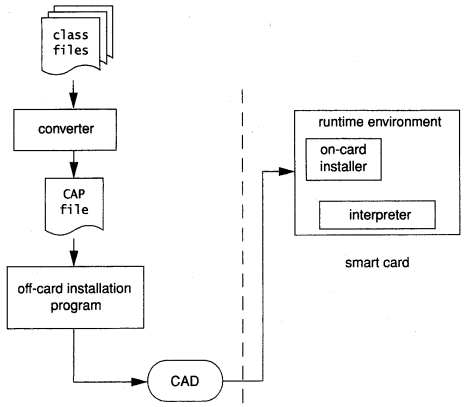


Compte tenu de la puissance de traitement d'une carte à puce la machine virtuelle est constituée par deux entités distinctes, l'une est logée sur une station de travail ou un ordinateur personnel (off-card VM), et l'autre est embarquée dans la carte (on-card VM).

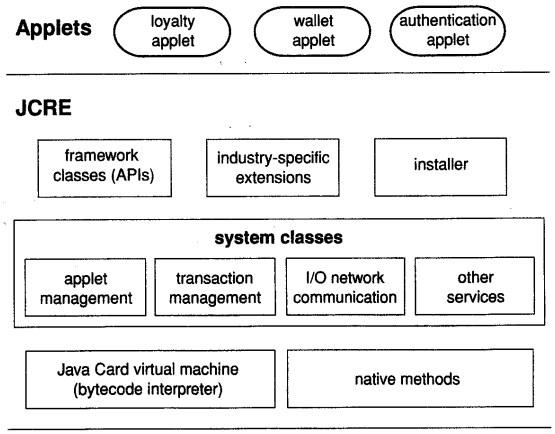


Un ensemble de fichiers java est compilé par un compilateur javac standard qui produit des fichiers .class. Un programme nommé *converter* implémente une partie de la machine virtuelle, il vérifie la comptabilité des classes avec le langage javacard, charge et lient ces objets dans un paquetage. Le résultat de ces opérations est stocké dans un fichier .*CAP* (Converted Applet). Un fichier dit d'exportation (.exp), contient les déclarations des différents éléments du paquetage afin d'en permettre une utilisation ultérieure par un autre paquetage, lors de l'opération de conversion.

Le fichier **.CAP** est chargé dans la carte grâce à la collaboration de deux entités le *off-card installation program* et le *on-card installer*. Cette opération consiste en la segmentation du fichier en une série d'APDUs qui sont généralement signés (et parfois chiffrés) à l'aide de clés secrètes afin d'authentifier leur origine.



Un interpréteur Java (*interpreter*) logé dans la carte réalise l'exécution du *code byte* lors de l'activation d'un Applet.



Le *Java Card Runtime Environnement* (JCRE) est un ensemble de composants résidants dans la carte.

- *Installer*, ce bloc réalise le chargement d'un Applet dans la carte.
- *APIs*, un ensemble de quatre paquetages nécessaires à l'exécution d'un Applet.
  - Java.lang (Object, Throwable, Exception).
  - javacard.framework.
  - javacard.security.
  - javacardx.security.
- Des composants spécifiques à une application particulière.
- Un ensemble de classes (*system classes*) qui réalisent les services de bases, tels que
  - Gestion des Applets
  - Gestion des transactions
  - Communications
  - Autres…
- La machine virtuelle et les méthodes natives associées.

Le dialogue avec l'entité JCRE est assuré à l'aide d'APDUs.

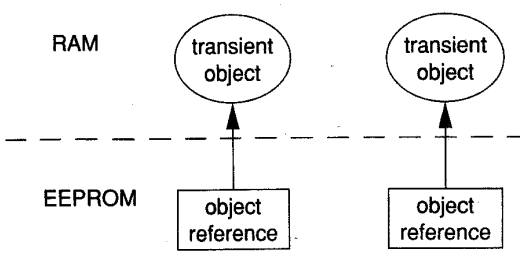
Cycle de développement d'un Applet.

Un Applet est un ensemble de classes regroupées dans un même fichier CAP. Il est identifié de manière unique par un AID (Application Identifier un nombre de 16 octets) qui conformément à la norme ISO7816-5 comporte un préfixe de 5 octets (RID – Resource IDentifier) et une extension propriétaire de 11 octets (PIX – Proprietary Identifier eXtension).

Un paquetage est également identifié par un AID, il est lié aux paquetages identiques (même AID) lors de son chargement.

L'environnement SUN comporte un émulateur de carte qui permet de tester le bon fonctionnement d'un Applet avant son chargement réel.

Les objets JavaCard.



En raison de la nature des cartes à puce le langage JavaCard supporte deux types d'objets, des objets de type persistant (*persistent object*) stockés dans la mémoire non volatile (E^2PROM), et des objets volatiles (*transient objets*) logés dans la mémoire RAM, dont l'image disparaît à chaque utilisation de la carte.

Par exemple

```
byte[] buffer = JCSystem.makeTransientByteArray(32,JCSystem.CLEAR_ON_DESELECT);
```

réalise la création d'un objet *transient*,détruit lors de la désélection de l'Applet.

Notion d'Atomicité.

Une carte à puce peut manquer d'énergie électrique lors de l'exécution d'un programme. La norme JC introduit une notion d'opération atomique relativement à la mémoire non volatile. Une procédure de transfert mémoire *atomique* est entièrement exécutée ou dans le cas contraire est sans effet.
De telles méthodes sont offerts par la classe javacard.framework.Util,

```
public static short ArrayCopy (byte [] src, short srcOffset, byte[] dest, short DestOffset, short length);
```

Notion de transaction.

Une transaction est une suite d'opérations qui modifient la mémoire non volatile. Cette suite de mises à jour est appliquée (*commit*) uniquement à la fin de la transaction, les modifications générés par une transaction sont appliquées de manière atomique ou ignorées.

Une méthode de la classe JCSystem réalise une telle transaction

```
JCSystem.beginTransaction() ;
// suite de modification dans la mémoire non volatile
JCSystem.commitTransaction() ;
```

Autres méthodes utiles
JCSystem.abortTransaction(), arrête une transaction en cours
JCSystem.getMaxCommitCapacity() , retourne le nombre maximum d'octets pouvant être mémorisés.
JCSystem.getUnusedCommitCapacity(), retourne le nombre d'octets encore disponibles.

Partage d'objets.

Un *contexte* JavaCard est un ensemble d'Applets qui appartiennent au même paquetage. Le système d'exploitation interdit toute interaction entre contextes pour des raisons évidentes de sécurité.
Cependant un tableau de type primitif (byte, short) avec le préfix *global* peut être accédé en lecture par différents contextes.

Afin de permettre à plusieurs applications d'échanger de l'information, comme par exemple la mise à jour de points de fidélité, on a introduit la notion d'interface partageable (*Shareable Interface Object* – SIO).

Un Applet dit *serveur* implémente un interface partageable. Un Applet dit *client* peut utiliser cette interface.

Exemple d'un Applet serveur

```
Public interface Miles extends Shareable
{public add(short x) ;}
public class AirMiles extends Applet implements Miles
{ static short count=0;
Public add(short x){ count += x ;}
// Méthode à surcharger pour autoriser le partage de l'interface
Public Shareable getShareableInterfaceObject(AID client_aid, byte parameter)
{// Vérification du paramètre AID
  return((Shareable)this;
}}
```

Exemple d'un Applet Client.

```
// Recherche de l'applet serveur identifié par son AID.
// public static AID lookupAID (byte[] buffer, short offset, byte length) ;
server_aid = lookupAID(buffer,0,(byte)16);
// Obtention d'une interface partageable
// public static Shareable getAppletShareableInterfaceObject(AID server_aid,byte parameter) ;
I = getAppletShareableInterfaceObject(server_aid,(byte)0); // retourne SIO ou null
I.add((short)5000) ;
```

Ressources cryptographiques.

L'accès aux fonctions cryptographiques telles que empreintes (*digest, md5,* sha1…), algorithmes à clés symétriques (DES, AES…) ou à clés asymétriques (RSA) est un point essentiel de la technologie JavaCard. Ces procédures sont écrites en code natif (assembleur, C) et généralement protégées contre les différentes attaques connues.
Le paquetage javacard.security comporte un ensemble d'interfaces et de classes, qui assurent le lien avec les ressources cryptographiques.

Digest.

On obtient un objet *MessageDigest* à l'aide de la méthode,

```
Public static MessageDigest getInstance(byte algorithm, boolean externalAccess);
```

Par exemple

```
MessageDigest md5 = MessageDigest.getInstance(MessageDigest.ALG_MD5,false) ;
MessageDigest sha  =  MessageDigest.getInstance(MessageDigest.ALG_SHA1,false) ;
```

Les méthodes et réalise les opérations usuelles des fonctions de *hash* telles mise à jour du calcul *update* et calcul final *doFinal*.

```
sha1.update(byte[] buffer, short offset, short length) ;
sha1.doFinal(byte[]buffer, short offset, short length) ;
```

Chiffrement.

La classe *KeyBuilder* permet de construire une clé (*interface Key*) non initialisée. Des interfaces spécifiques sont associées à chaque algorithme.

Exemples.

- DESKey des_key;
  des_key = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES, false) ;
- RSAPrivateKey rsa_private_key ;
  rsa_private_key =
  (RSAPrivateKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,
  KeyBuilder.LENGTH_RSA_512, false) ;

Une clé RSA est initialisée à l'aide de méthodes fournies par l'interface RSAPrivateKey

```
rsa_private_key.setExponent(byte[] buffer,short offset, short length) ;
rsa_private_key.setModulus(byte[] buffer,short offset, short length) ;
```

Une clé symétrique DES peut utiliser la méthode setKey de l'interface DESKey.

```
DesKey.setKey(byte[] KeyData, short offset)
```

Un objet Cipher réalise les opérations de chiffrement/déchiffrement.

```
Cipher cipher;
cipher = Cipher.getInstance(Cipher.ALG_DES_CBC_NO_PAD,false);
cipher.init((Key)des_key,cipher.MODE_ENCRYPT);
```

Les méthodes *update* et *doFinal* chiffrent les données.

```
cipher.update(byte[] inBuf, short inOffset, short inLength, byte[] outBuff, short outOffset);
cipher.doFinal(byte[] inBuf, short inOffset, short inLength, byte[] outBuff, short outOffset);
```

Signature.

La mise en œuvre d'une opération de signature est très similaire à celle d'un chiffrement.

On obtient une référence sur un objet signature,
Signature signature;
signature = Signature.getInstance(Signature.ALG_RSA_MD5_RFC2409,false);

La clé associée à la signature est initialisée par une méthode init
signature.init(Key thekey, byte theMode) ou
signature.init(Key thekey, byte theMode, byte[] bArray, short bOffset, short bLength);

Les opérations de signature utilisent les procédures
signature.update(byte[] buf, short offset, short length)
Signature.sign(byte[] buf, short offset, short length, byte[] sig_buf, short sig_offset);

Nombres aléatoires.

La classe *RandomData* réalise la génération de nombre aléatoire.

Exemple.

```
RandomData random_data ;
random_data = getInstance(RandomData.ALG_SECURE_RANDOM) ;
random_data.setSeed(byte[] seed, short offset, short length) ;
random_data.generateData(byte[] random, short offset, short length) ;
```

La classe Applet.

Une application (on emploie parfois le terme *cardlet*) est articulée autour de l'héritage d'une classe Applet.

Public class MyApplet extends Applet {} ;

La classe Applet comporte 8 méthodes.

- Static void install(byte[]bArray, short bOffset, byte bLength)
  Cette méthode est utilisée par l'entité JCRE lors de l'installation d'un applet. Le paramètre indique la valeur de l'AID associé à l'application. Les objets nécessaires à l'application sont crées lors de l'appel de cette procédure (à l'aide de l'opérateur *new*).
- Protected void register()
  Cette méthode réalise l'enregistrement de l'applet avec l'AID proposée par le JCRE.
- Protected void register(byte[]bArray, short bOffset, byte bLength)
  Cette méthode enregistre l'applet avec un AID spécifié dans les paramètres d'appel.
- Protected boolean select()
  Cette méthode est utilisée par le JCRE pour indiquer à l'applet qu'il a été sélectionné. Elle retourne la valeur *true*.
- Protected boolean selectingApplet()
  Indique si un APDU SELECT (CLA=A4) est associé à l'opération de sélection de l'applet.
- Protected void deselect()
  Cette méthode notifie la déselection de l'applet.
- Shareable getShareableIntercaeObject(AID clientAID, byte parameter)
  Cette méthode est surchagée par une application (serveur) qui désire offrir une interface partageable.
- Abstract void process(APDU apdu) throws ISOException
  Cette méthode est la clé de voûte d'un applet. Elle reçoit les commandes APDUs et construit les réponses. L'exception ISOException comporte une méthode ISOException.throwIt(short SW) utilisée pour retourner le status word (SW) d'un ordre entrant.

Le traitement des APDUs

La classe APDU comporte toutes les méthodes nécessaires à la réception des commandes et à la génération des réponses.
Les méthodes les plus employées sont les suivantes

- static byte getProtcol()
  Retourne le type du protocole de transport ISO7816 (0 ou 1). En règle générale seul le protocole T=0 est supporté.
- byte[] getBuffer()
  Cette méthode une référence sur un tampon de réception (situé en RAM). La taille de ce bloc est d'au moins 37 octets (5+32).

- Short setIncomingAnd Receive()
  Cette méthode retourne le paramètre Lc d'une commande entrante. Les Lc octets sont mémorisés à partir de l'adresse 5 du buffer (tampon) de réception.
- Void setOutgoingAndSend(short Offset, short Length)
  Cette méthode transmet un message de réponse contenu dans le tampon de réception. Dans le cas d'une APDU sortante (Lc=0 ET Le#0) le statut 9000 est ajouté à la fin du message. Dans le cas d'une APDU entrante et sortante (Lc#0 et Le#0) le mot de statut 61 Le est généré afin de permettre une lecture ultérieure par un GET_RESPONSE (CLA C0 00 00 Le).

Divers & utile.

- ISOException.throwIt(short SW), permet de quitter la méthode process en générant un mot de statu SW. La classe ISO7816 contient une liste de valeurs utiles.
- Les méthodes

Util.arrayCopy(byte[] Source,short OffsetSource,byte[] Destination,short OffsetDestadr, short Length;
ET
Util.arrayCopyNonAtomic(byte[] Source,short OffsetSource,
byte[] Destination,short OffsetDestadr, short Length;

Réalisent des transferts de mémoire à mémoire avec ou sans atomicité.

- Short Util.makeShort(byte MSB,byte LSB) fabrique un entier short à partir de deux octets.

Utilisation des outils SUN.

Un exemple d'Applet – DemoApp.

```
package Demo;
import javacard.framework.*;

public class DemoApp extends Applet
{
       final static byte  BINARY_WRITE = (byte) 0xD0       ;
       final static byte  BINARY_READ  = (byte) 0xB0       ;
       final static byte  SELECT       = (byte) 0xA4       ;
       final static short NVRSIZE      = (short)1024       ;
       static byte[] NVR               = new byte[NVRSIZE] ;

public void process(APDU apdu) throws ISOException
  { short adr,len;

  byte[] buffer = apdu.getBuffer() ; // lecture CLA INS P1 P2 P3

  byte cla = buffer[ISO7816.OFFSET_CLA];
  byte ins = buffer[ISO7816.OFFSET_INS];
  byte P1  = buffer[ISO7816.OFFSET_P1] ;
  byte P2  = buffer[ISO7816.OFFSET_P2] ;
  byte P3  = buffer[ISO7816.OFFSET_LC] ;

  adr = Util.makeShort(P1,P2) ;
  len = Util.makeShort((byte)0,P3)  ;

  switch (ins) {

              case SELECT: return;

              case BINARY_READ:
                     if (adr < (short)0)
                     ISOException.throwIt(ISO7816.SW_WRONG_LENGTH) ;
                     if ((short)(adr + len) > (short)NVRSIZE)
                            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH) ;
                     Util.arrayCopy(NVR,adr,buffer,(short)0,len);
                     apdu.setOutgoingAndSend((short)0,len);
                     break;

              case BINARY_WRITE:
                     short readCount = apdu.setIncomingAndReceive(); // offset=5 ...
                     if (readCount <= 0)ISOException.throwIt(ISO7816.SW_WRONG_LENGTH) ;
                     if (adr < (short)0)
                            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH) ;
                     if ((short)(adr + len) > (short)NVRSIZE )
                     ISOException.throwIt(ISO7816.SW_WRONG_LENGTH) ;
                     Util.arrayCopy(buffer,(short)5,NVR,adr,len);
                     break;

              default:
                     ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
              }
}

protected DemoApp(byte[] bArray,short bOffset,byte bLength)
{  register();
..//register(byte[] bArray,short bOffset,byte bLength);
  }

public static void install( byte[] bArray, short bOffset, byte bLength )
{  new DemoApp(bArray,bOffset,bLength); }

public boolean select() {return true;}

public void deselect(){}
}
```

Cahier des charges de DemoApp

Cet applet réalise des lectures et des écritures dans la mémoire non volatile (NVR).

- Package Demo
- Applet AID 0x4a:0x54:0x45:0x53:0x54:0x30:0x30
- Package AID 0x4a:0x54:0x45:0x53:0x54:0x30
- TPDUs supportées

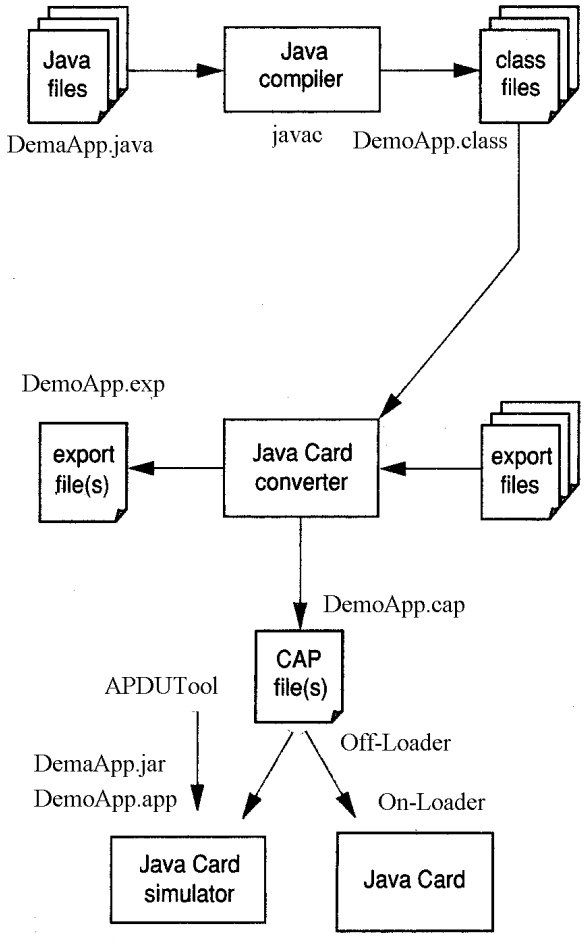
  Read 00 B0 P1 P2 Le.

  Write 00 D0 P1 P2 Lc [Lc octets].

  SelectAID 00 A4 04 00 07 4a 54 45 53 54 30 30.

Le répertoire racine des applets est localisé en c:\reader\Applet.

Le fichier java source est stocké en c:\reader\Applet\Demo\DemoApp.java



Compilation et conversion.

- Cette opération est réalisée par le fichier *batch* compile.bat. Les fichiers produits sont DemoApp.cap, DemoApp.exp et une fichier d'archive DemoApp.jar.

```
set JC=c:\jc211
set JDK=c:\JDK

set JCBIN=%JC%\bin
set CLASSPATH=%JCBIN%\api21.jar

set PACK=Demo
set APPLI=DemoApp

REM  compilateur javac
%JDK%\bin\javac.exe -classpath %CLASSPATH% -g %APPLI%.java

set PATH=%PATH%;%JCBIN%
set PATH=%PATH%;%JDK%

REM Converter converter.jar
%JDK%\bin\java -classpath %JCBIN%\converter.jar;%JDK%\lib com.sun.javacard.converter.Converter -config
%APPLI%.opt

cd ..
%JDK%\bin\jar.exe cvf %PACK%\%APPLI%.jar %PACK%\%APPLI%.class
cd %PACK%
```

- Le fichier DemoApp.opt mémorise la liste des options nécessaires au *converter*.

```
-classdir .\..
-exportpath C:\jc211\api21
-applet 0x4a:0x54:0x45:0x53:0x54:0x30:0x30 DemoApp
-out CAP EXP
-nobanner
Demo 0x4a:0x54:0x45:0x53:0x54:0x30 1.0
```

Emulation.

- Le fichier de commande emulator.bat

```
set JC=c:\jc211
set JDK=c:\JDK
set APPLI=DemoApp
REM
set JCBIN=%JC%\bin
REM
REM Emulateur JavaCard
%JDK%\bin\java -classpath %JC%\bin\jcwde.jar;%JC%\bin\apduio.jar;%JC%\bin\api21.jar;.\%APPLI%.jar
com.sun.javacard.jcwde.Main -p 9025 .\%APPLI%.app
```

Le fichier DemoApp.app

```
//              Applet                                          AID
com.sun.javacard.installer.InstallerApplet    0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0x8:0x1
Demo.DemoApp                                  0x4a:0x54:0x45:0x53:0x54:0x30:0x30
```

Test avec APDUtool.

- Le fichier apdutool.bat

```
set JC=c:\jc211
set JDK=c:\JDK
set FILE=script
REM
set JC21BIN=%JC%\bin
REM
set PATH=%PATH%;%JC21BIN%
set PATH=%PATH%;%JDK%

REM path
REM set
%JDK%\bin\java -noverify -classpath %JC21BIN%\apdutool.jar;%JC21BIN%\apduio.jar;%JDK%\lib
com.sun.javacard.apdutool.Main -p 9025 .\%FILE%.scr
```

- Le fichier d'apdu script.rc

```
powerup;
powerup;

// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
// 90 00 = SW_NO_ERROR

// begin installer command
0x80 0xB0 0x00 0x00 0x00 0x7F;
// 90 00 = SW_NO_ERROR

// create DemoApp applet
0x80 0xB8 0x00 0x00 0x09 0x07 0x4a 0x54 0x45 0x53 0x54 0x30 0x30 0x00 0x7F;
//                              AID=0x4a:0x54:0x45:0x53:0x54:0x30:0x30
// 90 00 = SW_NO_ERROR

//End installer command
0x80 0xBA 0x00 0x00 0x00 0x7F;
// 90 00= SW_NO_ERROR

// Select DemoApp
0x00 0xa4 0x04 0x00 0x07 0x4a 0x54 0x45 0x53 0x54 0x30 0x30  0x7F;
// 90 00 = SW_NO_ERROR

// Write 4 bytes
0xBC 0xD0 0x00 0x00 0x04 0x12 0x34 0x56 0x78 0x7F ;
//90 00

// Read 4 bytes
 0xBC 0xB0 0x00 0x00 0x00 0x04 ;
// 12 34 56 78 90 00

// *** SCRIPT END ***
powerdown;
```