Java Card™

Damien Sauveron
damien.sauveron@unilim.fr
http://damien.sauveron.fr/

Intervenants dans le module

Damien Sauveron (Cours, TD et TP) :

- Membre de l'équipe « Cryptis » en charge du projet « Sécurité des Systèmes et Réseaux » (P.F. Bonnefoi, E. Conchon, K. Tamine)
- Thème de recherche : sécurité des systèmes embarqués et des réseaux mobiles

Pour me contacter :

Damien Sauveron
XLIM UMR 7252 CNRS -- Université de Limoges
123 avenue Albert Thomas
87060 Limoges Cedex, FRANCE

Email: Damien.Sauveron@unilim.fr
Web: http://damien.sauveron.fr/
Phone: +33 (0) 5 87 50 67 93

MCF à l'Université de Limoges depuis septembre 2006 – HDR (2014)
Visite postdoctorale au Smart Card Centre de l'Information Security Group du Royal Holloway University of London
Thèse au LaBRI – Université Bordeaux 1 sur la Sécurité de la Technologie Java Card
Ingénieur R&D dans le CESTI de SERMA Technologies (Pessac)

Organisateur de WISTP2007 (Héraklion) et WISTP2008 (Séville)
Président du groupe de travail de l'IFIP : Pervasive Systems Security

Plan

Les concepts
- Qu'est-ce que Java Card ?
- Statut du standard Java Card
- Sous-ensemble du langage Java
- La machine virtuelle
- Le modèle mémoire

Les sécurités

Les attaques

La pratique
- Concepts de programmation
- Les APIs de programmation
- Développement d'une applet Java Card
- Construire une application avec la Java Card

État de l'art des cartes avant Java Card

La carte à puce est un véritable ordinateur utilisé comme serveur portable et sécurisé de données personnelles

Elle est programmée comme un composant embarqué avec un code applicatif figé

Elle est difficile à intégrer dans les systèmes d'informations
- Car si l'application requiert de nouvelles fonctions carte il faut développer un nouveau masque (=> coûteux)
- => Sinon il faut prévoir un masque qui accepte d'exécuter des programmes en EEPROM (rare et pas standardisé)

- Car pas d'interface standard de communication avec les différents lecteurs (travaux en cours – problème connexe)
  - Communication via APDUs (très bas niveau)

Problèmes à résoudre et/ou besoins à satisfaire
- Permettre le développement de programmes pour la carte sans avoir besoin de graver un nouveau masque
  - Faciliter et accélérer les développements de codes dans la carte
- Faire de la carte un environnement d'exécution de programmes ouvert (chargement dynamique de code)
  - Rendre plus souples et plus évolutives les applications carte
- Faciliter l'intégration des cartes dans les applications
  - Faciliter et accélérer les développements d'applications clientes des cartes

Besoins :
- Diversification des sources cartes et des terminaux
- Pérennité des développements

Procédé de construction des applications
- Permettre le développement «rapide» d'applications carte «évoluées» par des «non spécialistes»
- But : de nouvelles applications pour de nouveaux marchés !

Qu'est ce que Java Card ?

Technologie permettant de faire fonctionner des applications écrites en langage Java pour :

carte à puce

autres périphériques à mémoire limitée (par exemple iButton)

De manière intuitive :

Il s'agit d'une carte basée sur un interpréteur de bytecodes.

Carte à puce classique =

La technologie Java Card définit une plate-forme pour cartes à puce sécurisée, portable et multi-applicative qui incorpore beaucoup des avantages du langage Java.

Historique 1/2

En Novembre 1996, un groupe d'ingénieurs de Schlumberger cherche à simplifier la programmation des cartes à puces tout en préservant la sécurité (Cyberflex 1.0 au CNIT – Cartes'96).

==> la spécification Java Card 1.0

En Février 1997, Bull et Gemplus se joignent à Schlumberger pour cofonder le « Java Card Forum ».

Consortium de fabriquants :

- Cartes (Gemplus, Schlumberger, etc)
- Informatique : IBM, Sun
- Matériel : DEC, Motorola
- Utilisateurs : Banques (CitiBank, etc.)

Buts :

- promouvoir Java Card
- Faire des choix technologiques commums
- Contre pouvoir à Sun (officieusement)

Solutions :

- Un comité technique
- Un comité business

http://www.javacardforum.org/

En Novembre 1997, Sun présente les spécifications Java Card 2.0.

1998 : Cyberflex 2.0, GemXpresso, ...

Historique 2/2

En Mars 1999 sort la version 2.1 des spécifications Java Card. Elles consistent en trois spécifications :

The Java Card 2.1 API Specification.
The Java Card 2.1 Runtime Environment Specification.
The Java Card 2.1 Virtual Machine Specification.
Contribution la plus significative :

- Définition explicite de la machine virtuelle de la Java Card.
- Le format de chargement des applets.

En Mai 2000, sort une petite correction ==> version 2.1.1

En Octobre 2000, plus de 40 entreprises ont acquis la licence d'exploitation de la technologie Java Card.

En juin 2002, spécifications Java Card 2.2.
Introduction de RMI, des canaux logiques, de nouveaux algorithmes cryptographiques, ...

En octobre 2003, spécifications Java Card 2.2.1.

En novembre 2005, la version 2.2.2 est Public Review Draft

En décembre 2005, les licenciés sont :
ARM, Aspects, Axalto, CCL/ITRI, Fujitsu, Gemplus, Giesecke & Devrient GmbH, GoldKey Technology, HiSmarTech, I'M Technologies Ltd., IBM, incard, KEBTechnology, Logos Smart Card, Microelectrónica Española, Oberthur Card Systems, ORGA Kartensysteme, SAGEM, Sermepa, Setec, Sharp, Smart Card Laboratory Inc., SSP Solutions, Inc., STMicroelectronics, Toppan Printing, Trusted Logic. + d'autres qui préfèrent rester anonymes.

JavaCard Forum Members

(as of 10th Jan 2005)

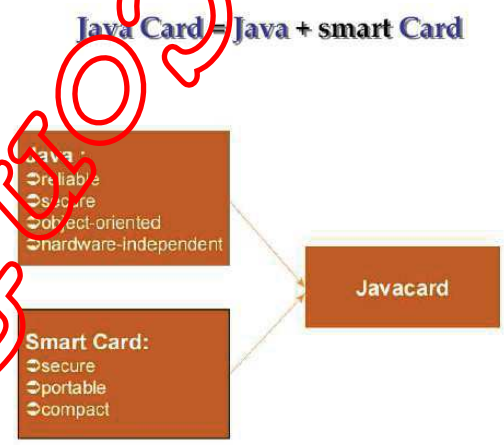
Les avantages de la technologie Java Card

La facilité de développement des applications} grâce :
à la programmation orientée objet offerte par Java
à l'utilisation des environnements de développement existants pour Java.
à une plate-forme ouverte qui définit des APIs et un environnement d'exécution standard.
à l'encapsulation de la complexité sous-jacente du système des cartes à puce.

L'indépendance au hardware réalisée grâce au langage Java
==> « *Write Once, Run Anywhere* »

La gestion dynamique de multiples applications
==> possibilité de mises à jour des applications de la Java Card sans avoir besoin de changer de cartes
==> possibilité d'offrir des nouveaux services

La compatibilité avec les standards existants pour les cartes à puce
IS07816 :
- Protocole de communication
- Protocole de haut niveau pour la gestion de fichiers, etc



On verra les inconvénients plus tard.

La plate forme Java

- Java est un langage de programmation

Voir le « white paper » de J.Gosling

Un programme Java est compilé et interprété

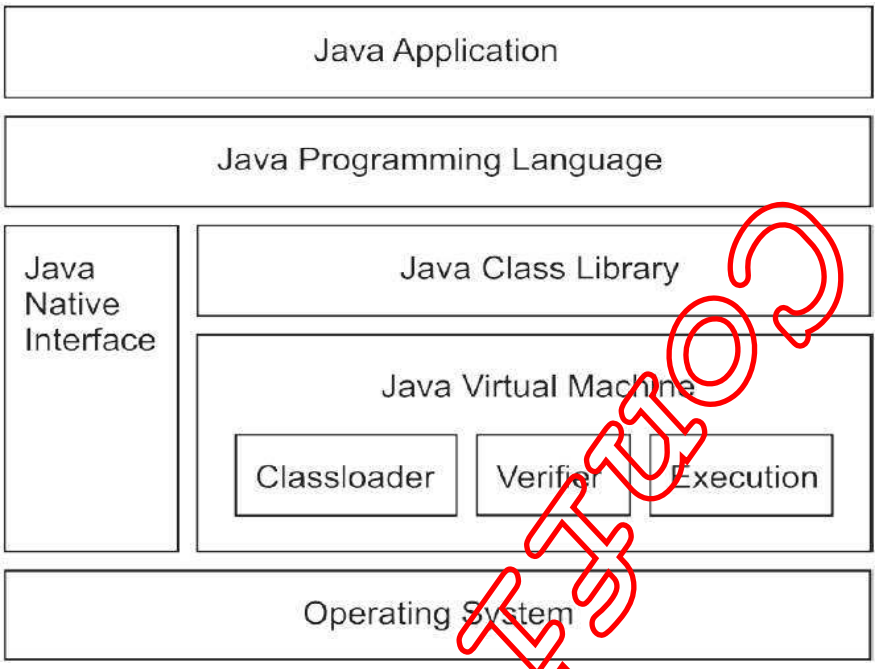
- Java est une plate-forme

La plate-forme Java, uniquement software, est exécutée sur la plate-forme du système d'exploitation,

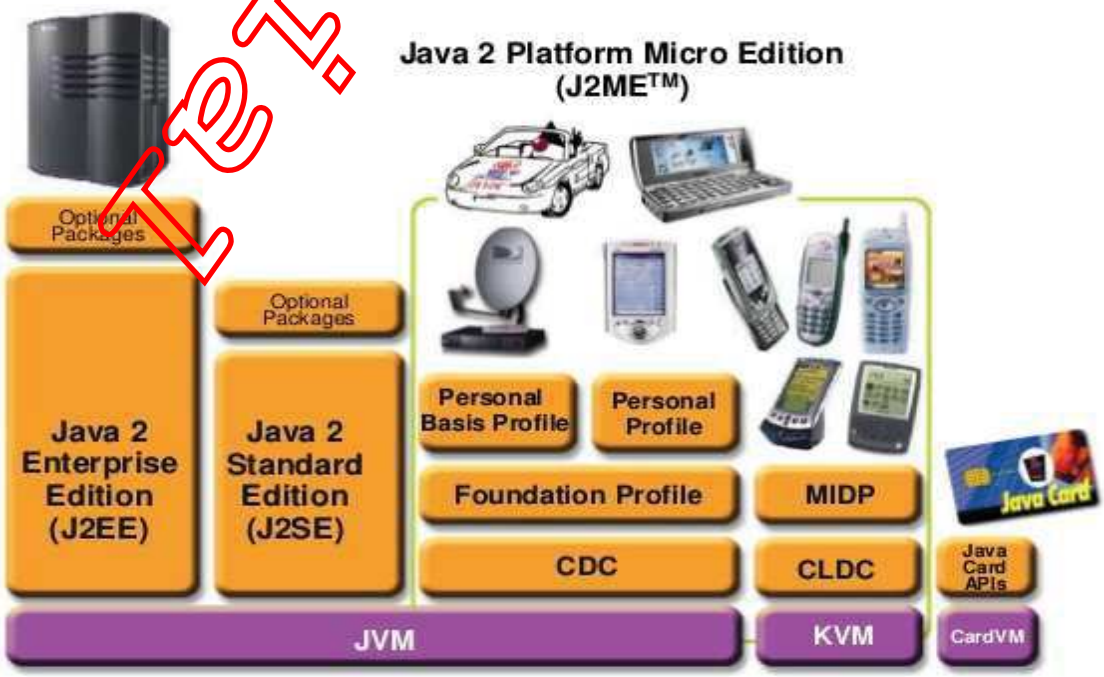
La « Java Platform » est constituée de:

- La « Java Virtual Machine » (JVM)
- Des interfaces de programmation d'application (Java API),
- Une interface pour les méthodes natives (JNI)

Architecture de la plate-forme



Les différentes plates-formes



Les composants

- Le langage de programmation,
- La librairie,
- La machine virtuelle

Un jeu d'instruction interprétable (*byte codes*)

Un format de chargement, pas forcément équivalent au format d'exécution le *class file*

Un algorithme de vérification des règles de typage Java ramenées au byte code, obligatoire pour les applets,

Un compilateur à la volée.

Java langage de programmation

- Java est un langage de programmation particulier qui possède des caractéristiques avantageuses:

Simplicité et productivité:

- Intégration complète de l'OO
- Gestion mémoire (« Garbage collector »)

Robustesse, fiabilité et sécurité (langage fortement typé)

Indépendance par rapport aux plates-formes

Distribution et aspects dynamiques

Performance

Machine virtuelle Java

- La JVM définit:

Les instructions du CPU

Les différents registres (PC, SP,…)

Le format des fichiers .class

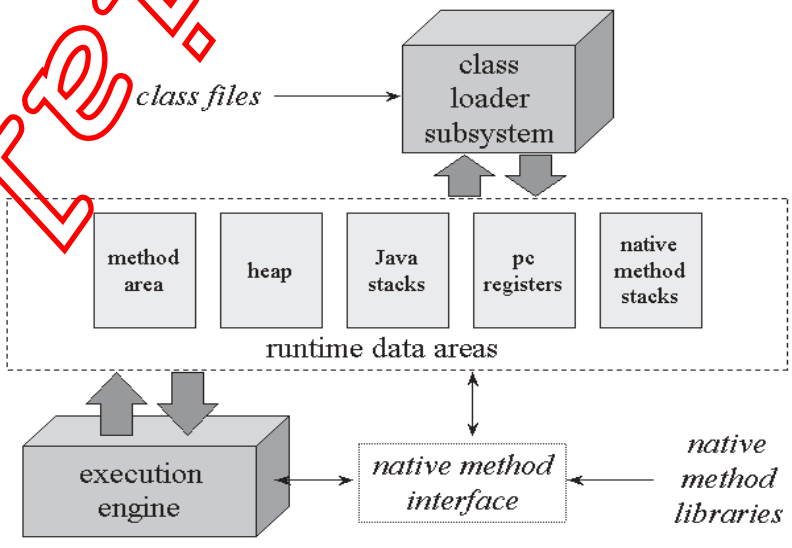
La pile Java et la pile C, en RAM,

Le tas (Heap) des objets « garbage-collectés » en RAM aussi,

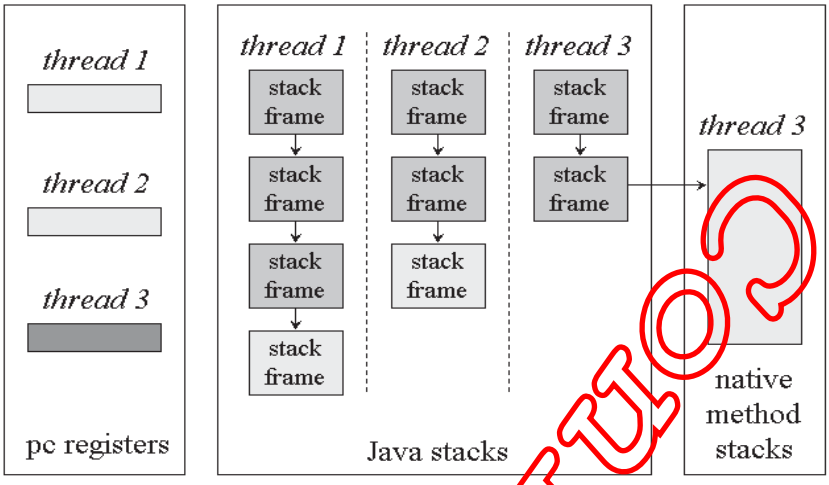
L'espace mémoire partagé:

- Stockage des classes et des données statiques.
- Stockage des objets (les instances) dans le tas.

Machine virtuelle Java



Machine virtuelle Java



Composition de la frame Java

- Elle contient:

Les variables locales définies à la compilation.

La pile des opérandes pour l'interprétation,

Les données de la frame.

- Résolution du constant pool,
- Retour normal d'exécution,
- Répartiteur d'exception.

La frame: les variables locales

```
class Example3a {

    public static int runClassMethod(int i, long l, float f, double d,
                                     Object o, byte b) {
        return 0;
    }
    public int runInstanceMethod(char c, double d, short s, boolean b) {
        return 0;
    }
}
```

runClassMethod()

index	type	parameter
0	int	int i
1	long	long l
3	float	float f
4	double	double d
6	reference	Object o
7	int	byte b

runInstanceMethod()

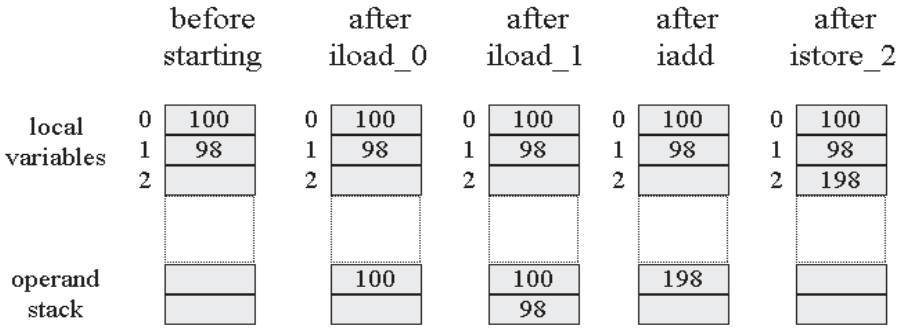
index	type	parameter
0	reference	hidden this
1	int	char c
2	double	double d
4	int	short s
5	int	boolean b

La frame: la pile d'opérandes

Addition de 2 nombres

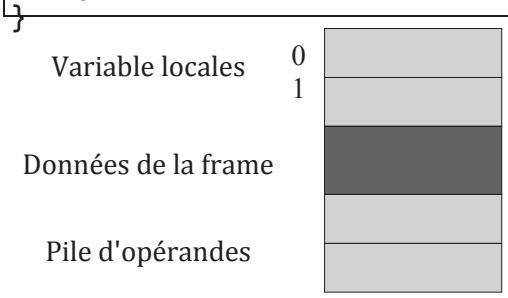
```
iload_0     // push the int in local variable 0
iload_1     // push the int in local variable 1
iadd        // pop two ints, add them, push result
istore_2    // pop int, store into local variable 2
```

Le compilateur a défini statiquement la taille de la pile pour chaque méthode



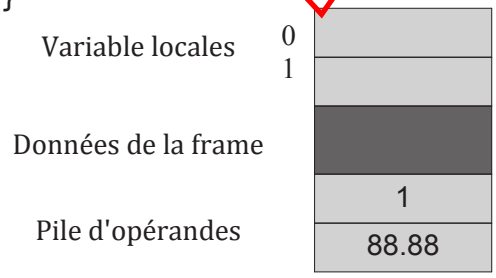
Empilement des frames

```
class Example3c {
    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```



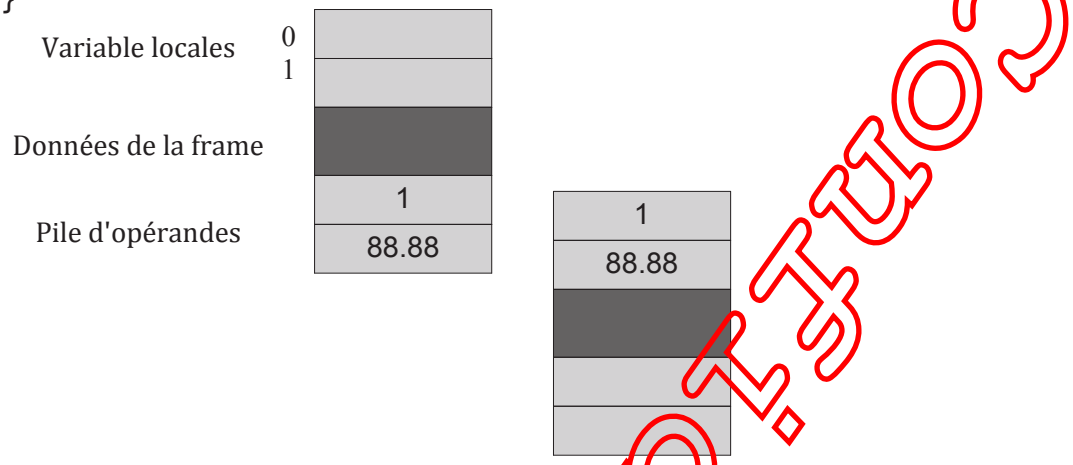
Empilement des frames

```
class Example3c {
    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```



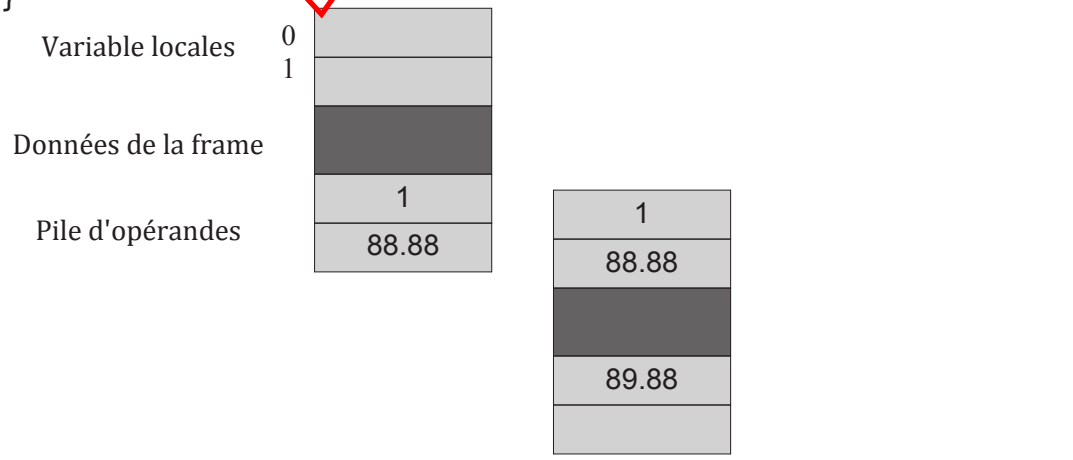
Empilement des frames

```
class Example3c {
    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```



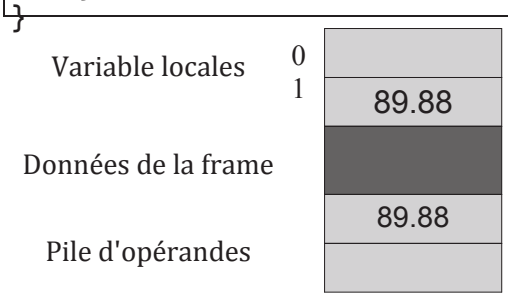
Empilement des frames

```
class Example3c {
    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```



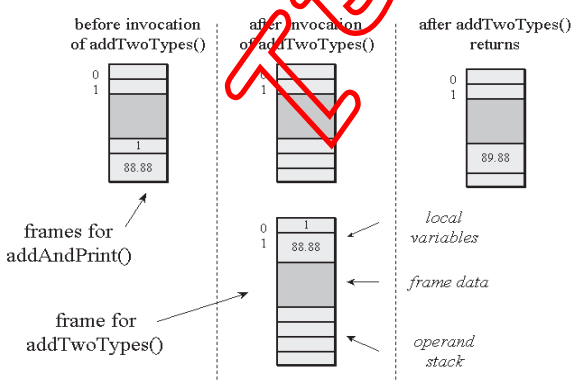
Empilement des frames

```
class Example3c {
    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```

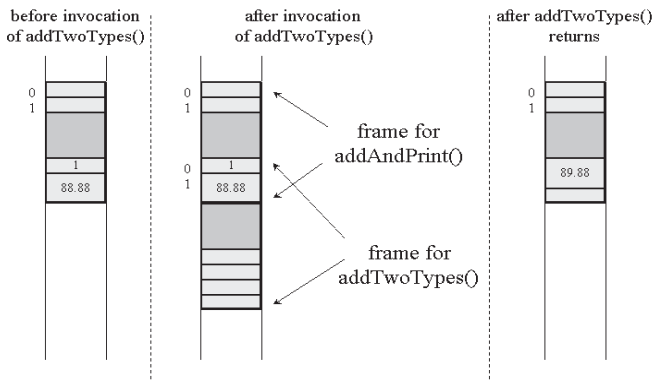


Allocation des frames

- Depuis un tas



- Depuis une pile contigu

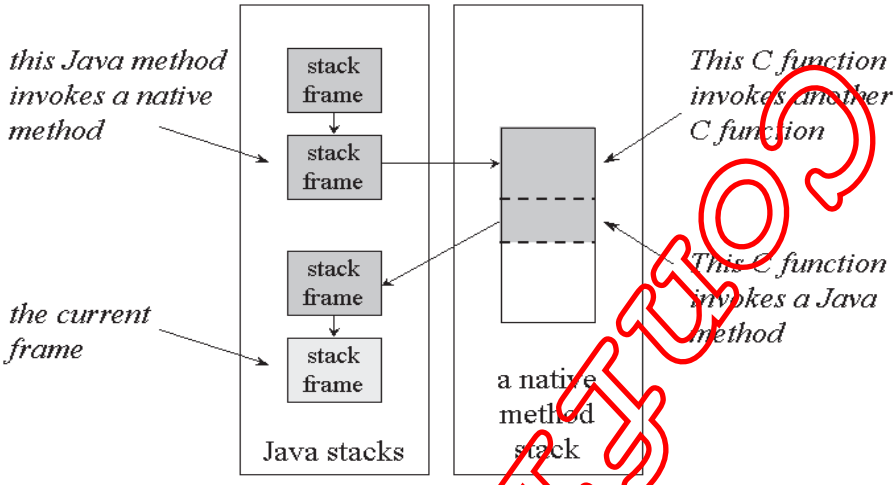


Un peu plus rapide car pas de copie des paramètres à l'invocation

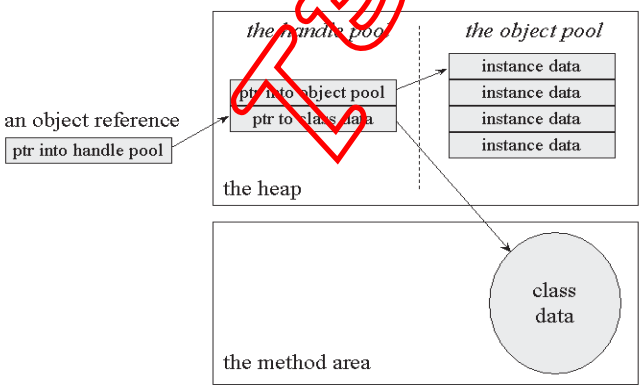
Une approche mixte est aussi possible.

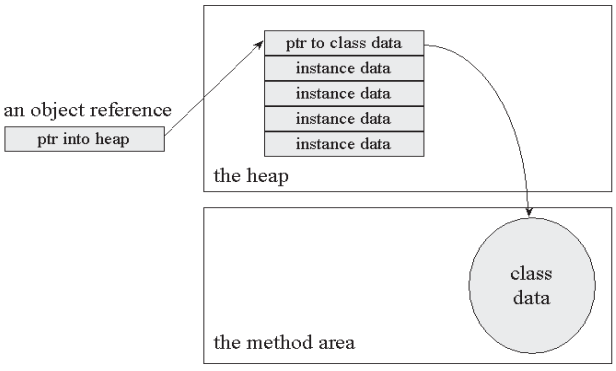
La frame: appel aux méthodes natives

- Une pile pour le langage natif est gérée par le système pour les appels



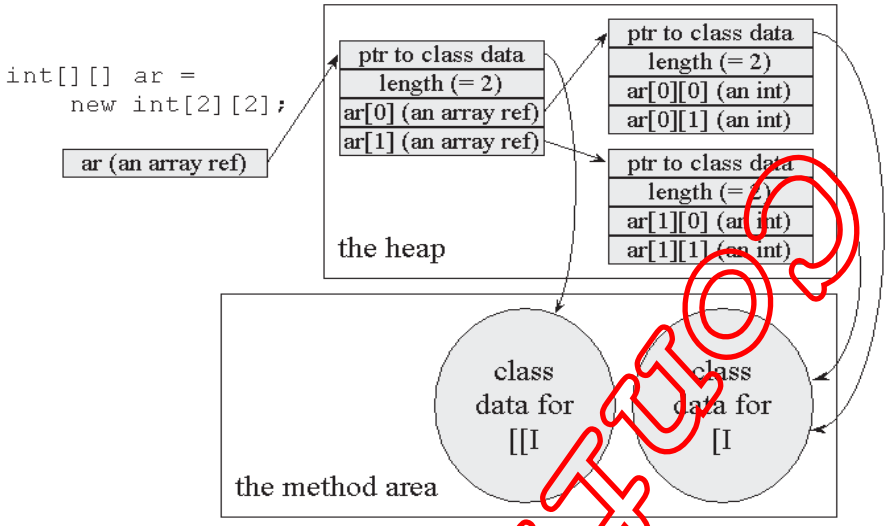
Le tas: instances & tableaux





Plus facile de gérer la fragmentation mais nécessite la résolution de 2 pointeurs !

Le tas: instances & tableaux



Machine Virtuelle Java

- Trois tâches principales:

Charger le code (class loader)

Vérifier le code (byte code verifier)

Exécuter le code (runtime interpreter)

- D'autres threads s'exécutent:

Garbage collector

(JIT compiler)

Sécurité par le typage

- En Java les pointeurs existent mais:

la manipulation explicite d'un pointeur en tant que tel est impossible en Java le compilateur vérifie cette règle ainsi que la VM. Ainsi le code suivant n'est pas compilable:

```
…
p=new Object();
p++;
```

tous les objets Java sont en fait des pointeurs (que l'on nomme références).

- Les variables locales (variables de types primitifs et références) sont allouées sur la pile (ou dans les registres) et les objets dans le tas.
- On ne contrôle pas la zone d'allocation ni quand sera effectuée la désallocation.

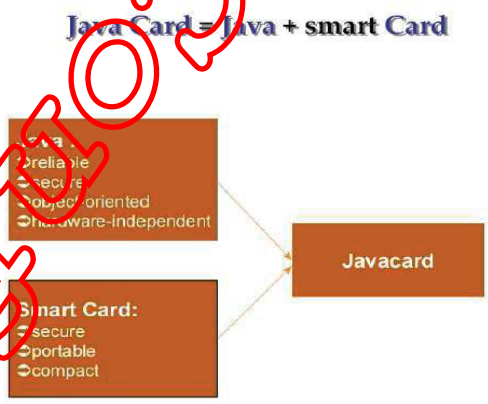
Sécurité par le typage

- Lors du chargement la VM vérifie si, au niveau byte code, il n'y a pas de transtypage ne correspondant pas aux règles de Java : c'est le rôle du vérifieur de type.
- Après, une référence n'a PLUS besoin de vérifier si le typage est correct.
- Donc la sécurité par le typage repose sur:

Un algorithme dans le compilateur,

Un autre dans la VM.

Java Card



Attention

- Les informations concernent Java Card 3.0 Classic Edition (pas Connected Edition) et Java Card 2.X

Présentation générale

- Un ensemble de spécifications

Publiées par Oracle

Basées sur la plateforme Java

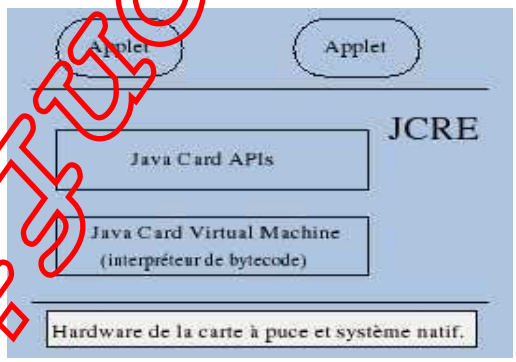
http://www.oracle.com/technetwork/java/embedded/javacard/index.html

- Découpées en trois parties

Application programming interfaces (APIs)

Execution environment (JCRE)

Virtual machine (VM)



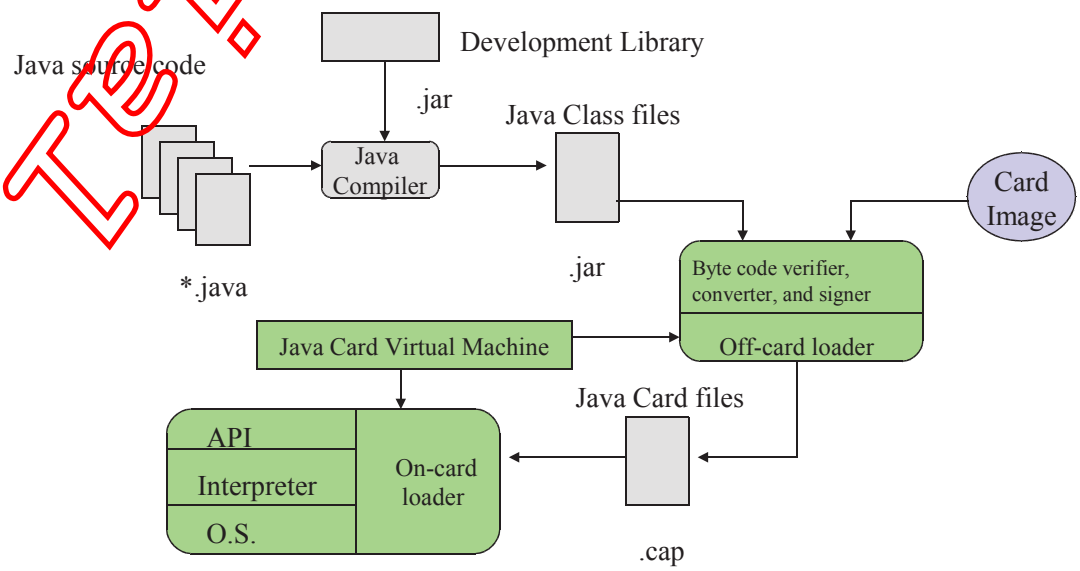
Agenda

- Introduction
- JCRE (Runtime Environment)
- JCVM (Virtual Machine)
- APIs

Un autre point de vue

- Architecture
- Mémoire
- Les différentes versions

L'architecture Java Card



Architecture

- Converter (part of the JCDK):

Class file format takes too much space on Java Card

Produces a format that fits the Smart Card constraints,

- Tokenization of the format

Need a representation of the content for pre-linking

The converter uses all class files of a package and all export files of ALL imported packages,

Output an Export file and a Cap file

- Conversion process

Verifies the Java Card language restriction

Optimize byte code

Invokes the off card verifier

Two specific file formats

- The CAP (Converted Applet) file format

Contains all the classes from one package

Semantically, is equivalent to a set of class (.class) files

Syntactically, differs a lot from class (.class) files

– All "string names" are replaced by "token identifiers"

- The EXP (Export) file format

Maintains the consistency between the originated class (.class) files and the resulting CAP file

– Only for public (exported) data

– Contains API information for a package of classes (access scope, class name, method signature, …)

Can be freely distributed, used during pre-linking phase

Not loaded into the card

The CAP file

- Contains an executable representation of package classes
- Contains a set of components (11)
- Each component describes an aspect of CAP file

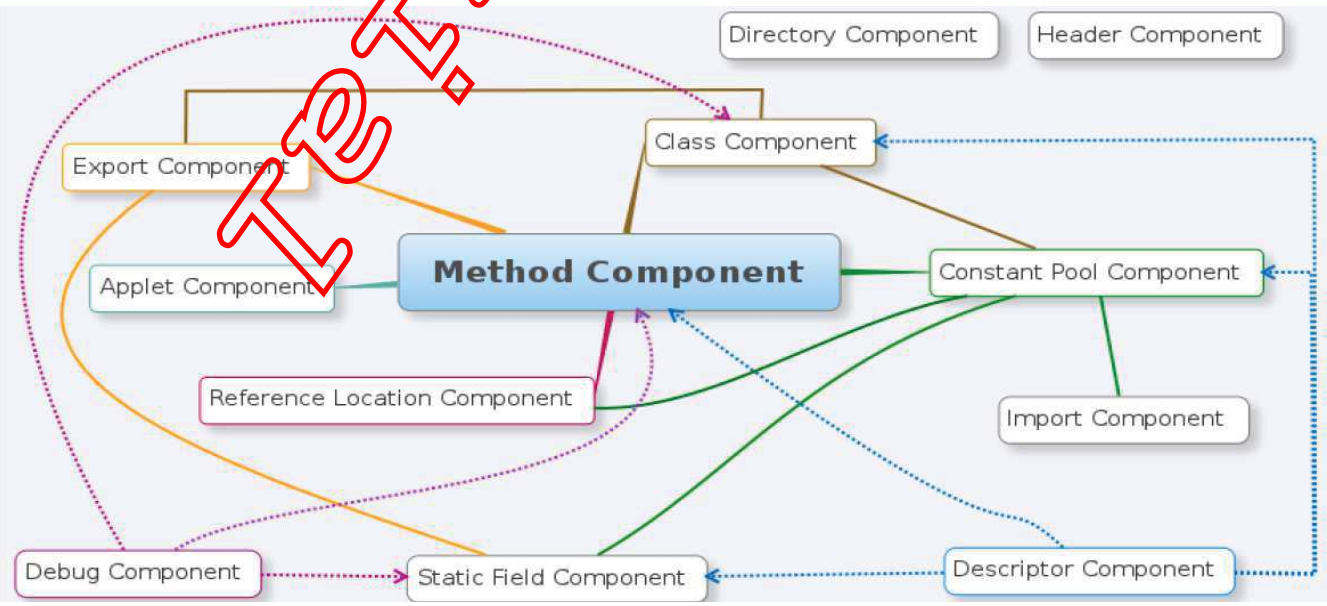
Class info

Executable byte code

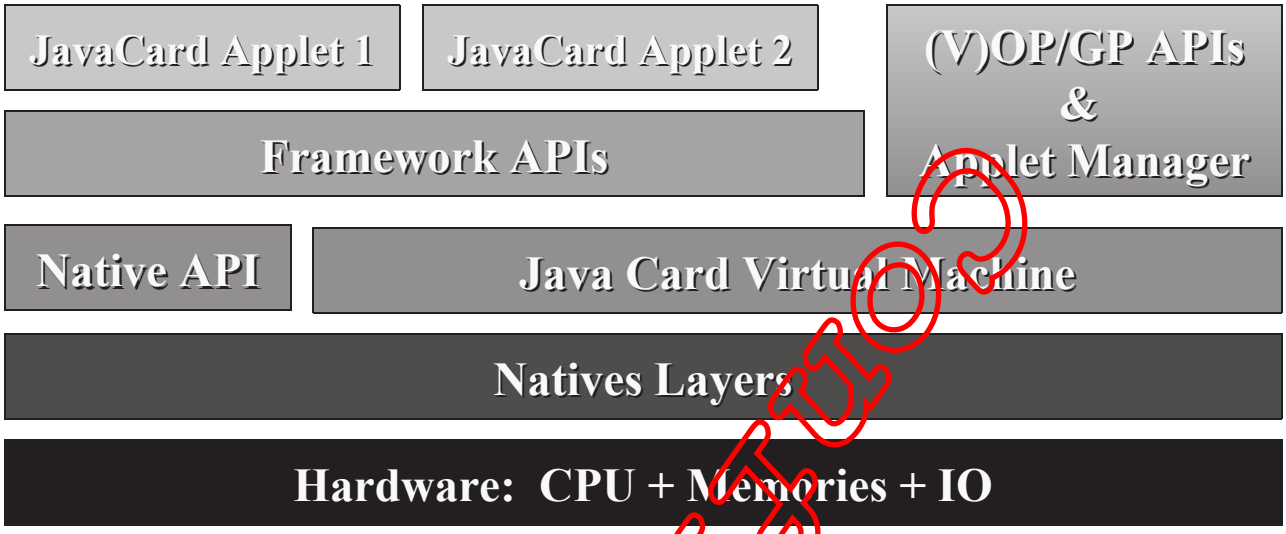
Linking info,…

- Optimized for small footprint by compact data structure
- Loaded on card

Interdependences



Java Card architecture



Java Card memory model

- By default, all objects are *implicitly* persistent

Because we have few RAM

Objects must survive between two sessions

- Some arrays can be *transient*

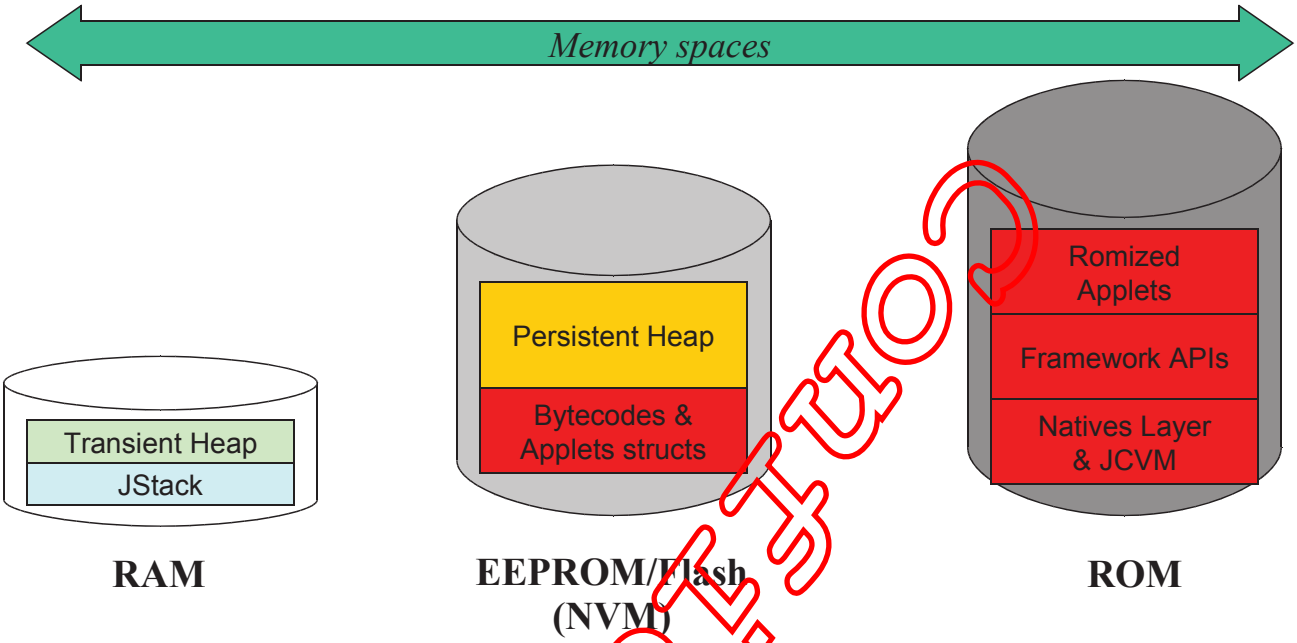
For efficiency and security reasons

- Transactional mechanisms are provided

All write operations on persistent memory are atomic

At the programming level a mechanism to handle transactions is also available

Java Card memory



Java Card 2.1

- File format of applet:

  Standardized

  file .cap identical to class excepted:

  - One .cap file per Java package

- « Firewall » between applets

  The virtual machine must ensure that the code does not run out of its execution space (context)

  Means to switch the execution context

  - Entry point object and global array can be accessed by applets (e.g., APDU)
  - JCRE can access each object
  - Interaction between applet through a shareable interface
  - System.share (Object ...) method suppressed

Java Card 2.2

- Load process of the .cap file standardized

Interoperability of JC 2.1 stops at the smart card loading,

- Object, applet and package deletion
- On card verifier (optional),
- Logical channels,
- Optional Garbage collector on demand,
- Support elliptic curves and AES algorithm,
- JC-RMI

Skeleton/stub generator 'a la RMI' hide the APDU encoding-decoding

- Rapid development and integration of SC applets

Agenda

- Introduction
- JCRE (Runtime Environment)
- JCVM (Virtual Machine)
- APIs

Execution environment (JCRE)

- Define how a Java Card manages its resources
- Define constraints on the Java Card operating system

Applet lifetime (installation, register and deletion)

Logical channels and applet selection,

Transient objects,

Applet isolation (firewall) and sharing,

Transaction and atomicity,

- RMI
- The JCRE is at the heart of a Java Card

APDU commands

- 2 types of APDU can be sent to the card:

OS/Administrative commands

– OS commands available in JCRE and CM

`Select, Load, Install …`

– Administrative commands specified by SC manufacturer

`Get Info, ...`

Applicative commands

– specific to the JC applets loaded in the card

– eg: `debit, credit, getbalance` for an e-purse applet

– eg: `create file, update file` for the GSM applet

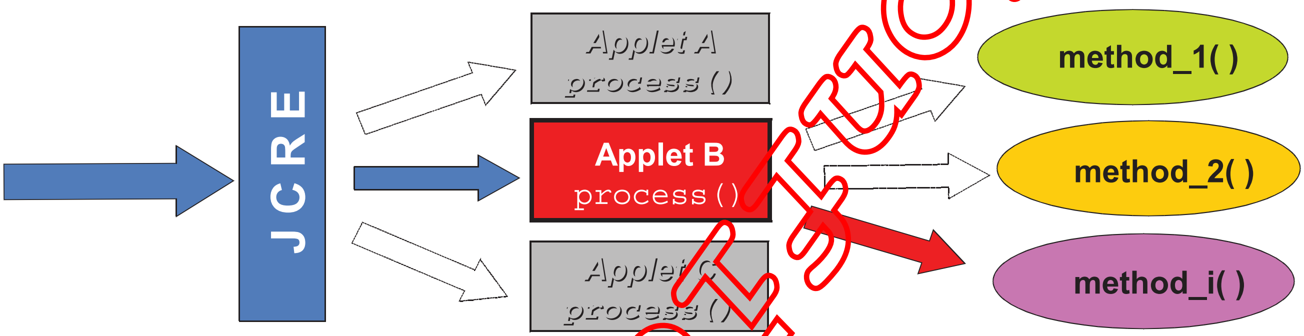
2 command dispatchers

- JCRE's task: main dispatcher

Route the incoming commands to the JCVM and the selected applet

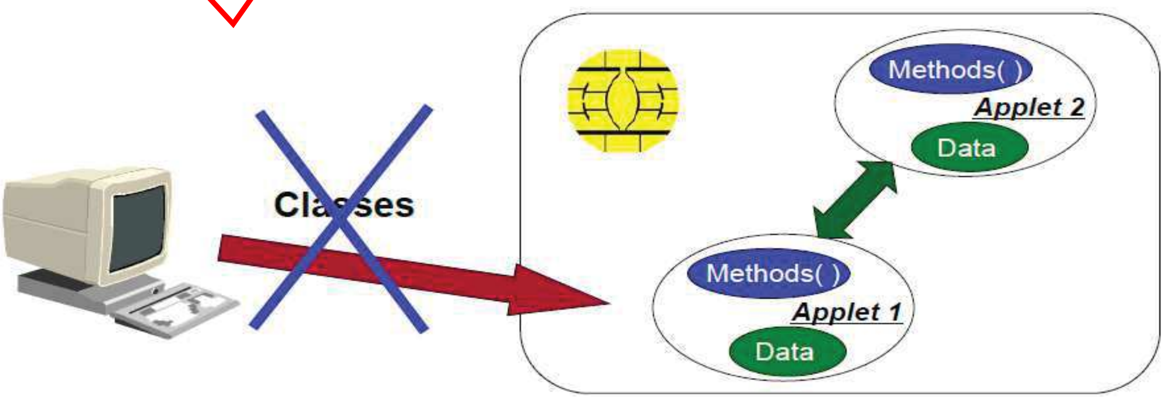
- Applet designer's task: second dispatcher

Implement the applet's command dispatcher (extraction of the header information and call of the associated method)

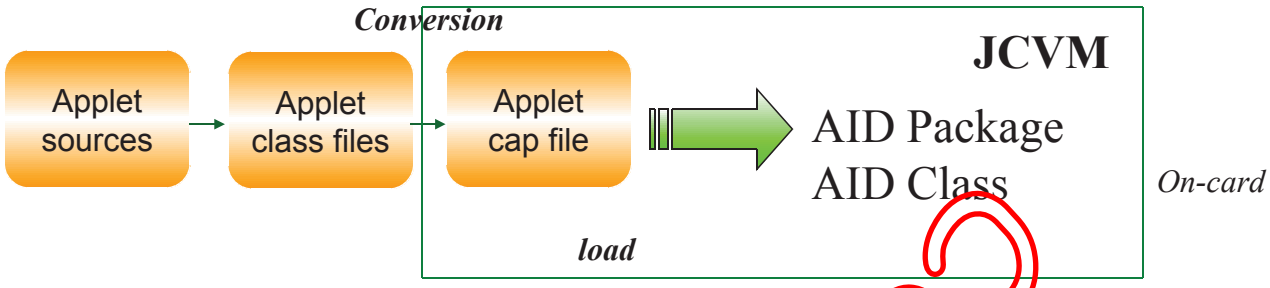


Unsupported features Dynamic class loading

- Classes are statically linked before being downloaded
- No way to download classes on the fly as needed...
- Applets only refer to classes which already exist on the card

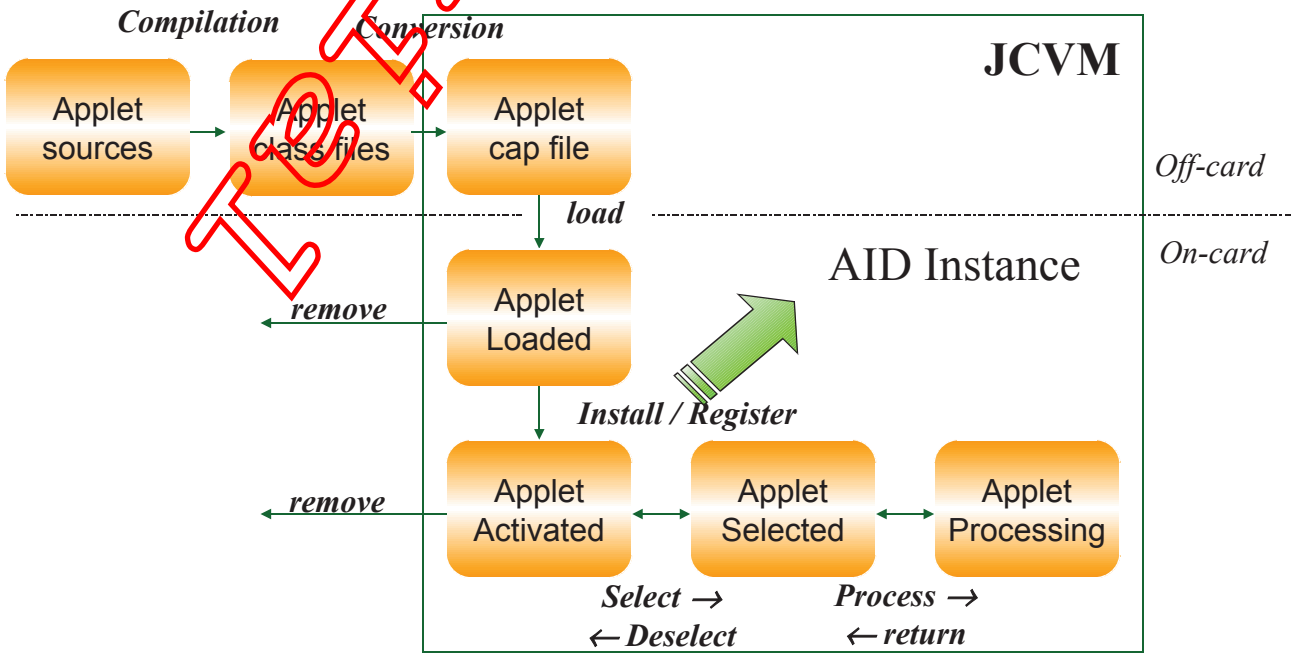


Java Card applet life cycle



Un AID : RID (5 octets) | PIX (0-11 octets)

Java Card applet life cycle



Un AID : RID (5 octets) | PIX (0-11 octets)

Methods of the applet

- An applet extends `javacard.framework.Applet`,
- It MUST implement the method `install`
- It MUST call the method `register`
- It MUST implement the method `process`
- It SHOULD implement `select, deselect`
- The `process` method can only be called if the applet has been selected

`install` method

- JCRE calls this static method first and gives

Applet instance AID

Applet privilege

Applet parameters

- `install` method creates an instance of an `Applet` subclass

Performs any necessary initializations,

Must call once one of the `register` methods

If no parameter is provided, only one installation

`register` method

- Used by the applet to register this applet instance
- Interacts with the Java Card Runtime Environment

`final void register()`

- Assign the applet instance AID with default AID bytes defined in CAP file

`final void register(byte [] bArray, short bOffset, byte bLenght)`

- Assign applet instance AID with the specified AID bytes

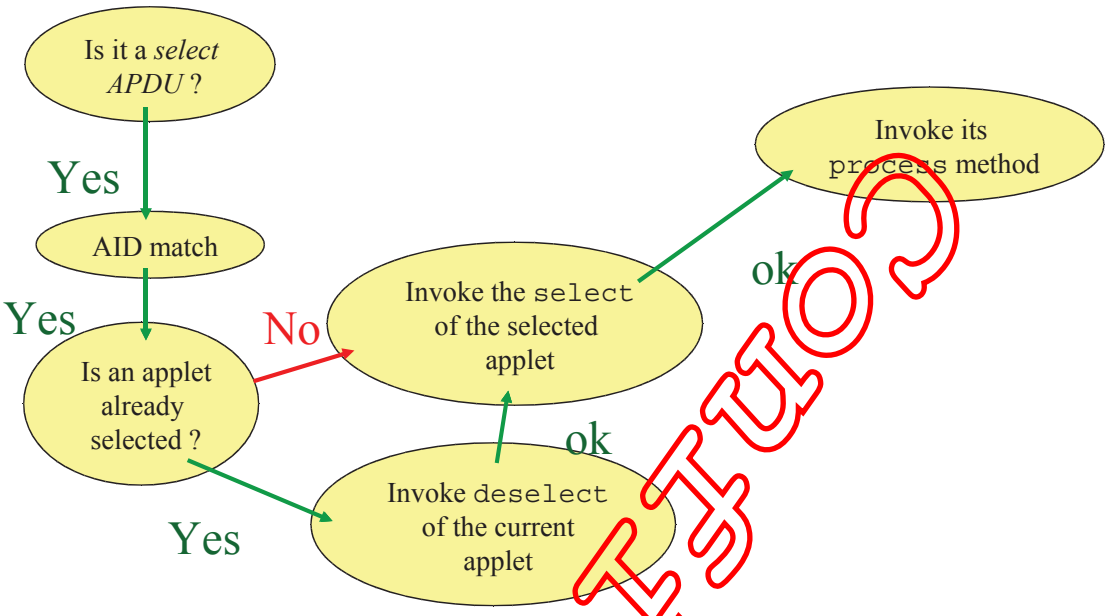
- Warning: when receiving the byte array as parameters of the install command, the length is sent before the AID

```
public static void install(byte[] bArray, short bOffset, byte bLength)
throws ISOException {
=>
register(bArray, (short)(bOffset + 1), (byte)bArray[bOffset]);
```

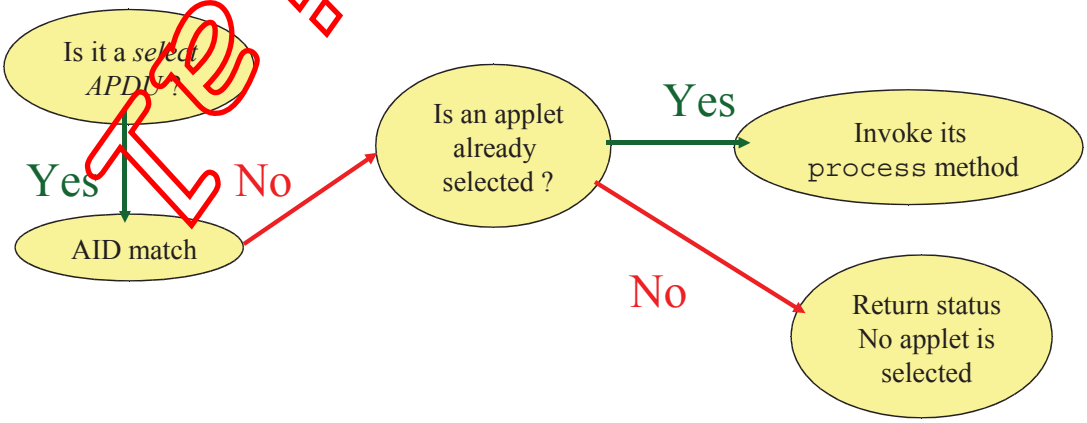
`process` method

- Contains the core application code of the applet
- Handles all the incoming APDU messages for the applet selected
- Called by the JCRE
- Upon normal return from this method the Java Card runtime environment sends the ISO 7816-4 success status word 90 00
- If it throws an exception the JCRE sends the associated reason code as the response status

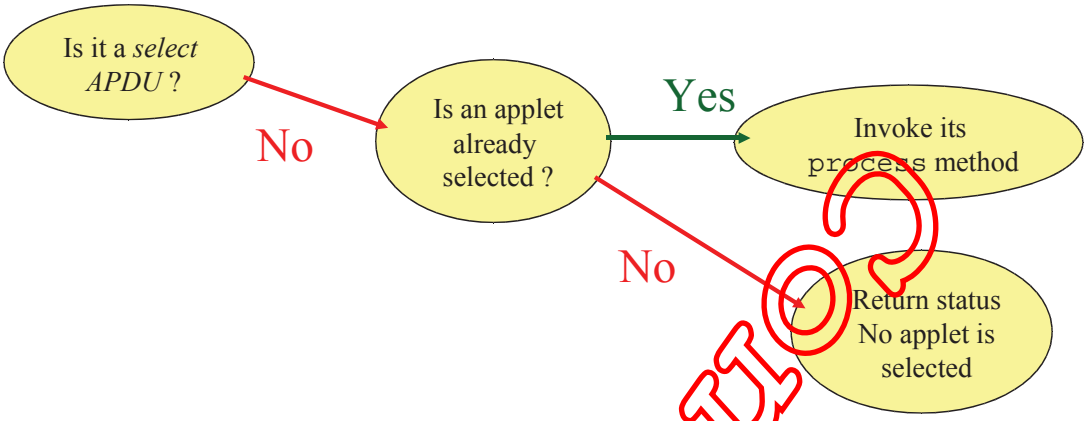
Selection protocol



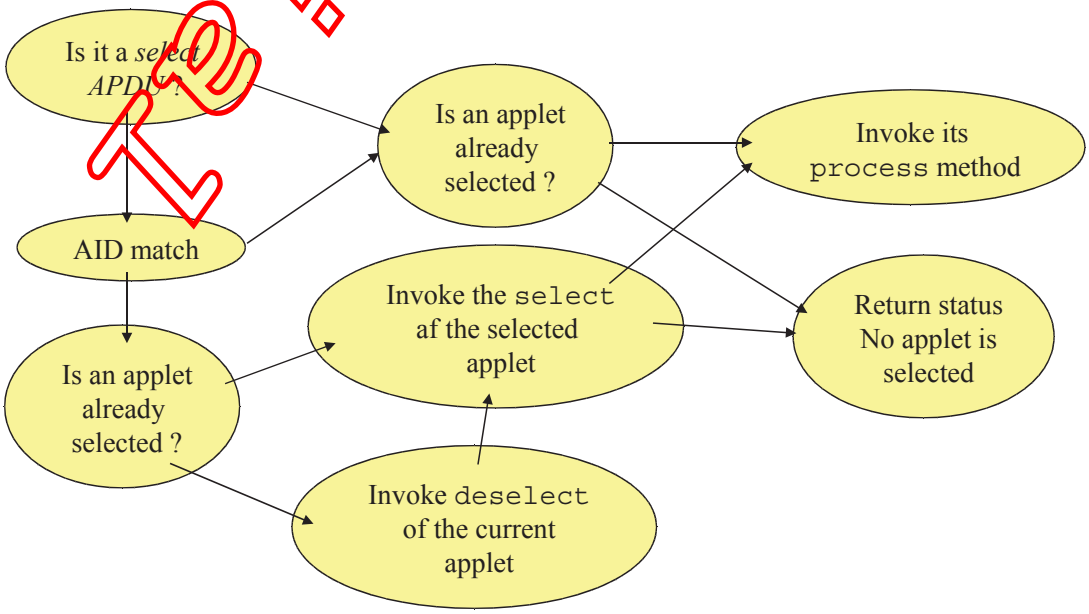
Selection protocol



Selection protocol



Selection protocol



Agenda

- Introduction
- JCRE (Runtime Environment)
- JCVM (Virtual Machine)
- APIs

Virtual machine (VM)

- This specification is made of several parts

The definition of the Java language subset that is supported

The definition of 2 specific file formats

The definition of a specific instructions set

Java language subset

- Supported
  - boolean, byte, short
  - int (optional)
  - Objects
  - Arrays (max size: 2^{15})
  - Virtual methods
  - Dynamic allocation
  - Packages
  - Exceptions
  - Interfaces
- Not supported
  - float, double, long
  - char, String
  - Multi-dimensional arrays
  - Garbage collector
  - Finalization
  - Threads
  - Dynamic loading of classes
  - Security manager

BC interpretation

- It is the execution engine for the byte code loaded into the card,
- It controls byte code execution, memory allocation and participate to the security through the firewall,
- Often it includes more tests than the firewall due to the absence of BC verifier…

Agenda

- Introduction
- JCRE (Runtime Environment)
- JCVM (Virtual Machine)
- APIs

APIs (1/2)

- Define a set of Java classes

  Used to develop Java Card services (applets)

  Dedicated for a smart card environment
- Those classes comply with current standard

  Business Java (`java.lang.Object`)

  ISO-7816 (*e.g.*, APDUs)

  Cryptographic
- Those classes do not define by themselves services

APIs (2/2)

- Contains 3 mandatory packages

`java.lang`

- Basic classes of the language

`javacard.framework`

- Framework of classes and interfaces for the core functionality of an applet (AID, APDU, Applet, ISO, PIN, JCSystem, Util, and exceptions classes)

`javacard.security`

- Core classes dedicated to cryptographic services (public/private key, random number generator,…)

- And one optional package

`javacardx.crypto`

- Implementation classes for ciphering/deciphering (strong cipher)

java.lang

- `Object { public Object();`
  `public boolean equals(Object obj); }`

- `Throwable { public Throwable(); }`
  `-- Exception`
  `-- RuntimeException`
  `-- ArithmeticException`
  `-- ClassCastException`
  `-- NullPointerException`
  `-- SecurityException`
  `-- ArrayStoreException`
  `-- NegativeArraySizeException`
  `-- IndexOutOfBoundsException`
  `-- ArrayIndexOutOfBoundsException`

javacard.framework

- class java.lang.Object
  - class javacard.framework.**AID**
  - class javacard.framework.**APDU**
  - class javacard.framework.**Applet**
  - class javacard.framework.**JCSystem**
  - class javacard.framework.**OwnerPIN** (implements javacard.framework.**PIN**)
  - class java.lang.Throwable
    - class java.lang.**Exception**
      - class javacard.framework.**CardException**
        - class javacard.framework.**UserException**
      - class java.lang.**RuntimeException**
        - class javacard.framework.**CardRuntimeException**
      - class javacard.framework.**APDUException**
      - class javacard.framework.**ISOException**
      - class javacard.framework.**PINException**
      - class javacard.framework.**SystemException**
      - class javacard.framework.**TransactionException**
    - class javacard.framework.**Util**
  - interface javacard.framework.**ISO7816**
  - interface javacard.framework.**PIN**
  - interface javacard.framework.**Shareable**

} Optionally used by the developper

- class javacard.framework.**ISOException** used very often in applet development

- **public class AID**

  Encapsulates constants related to ISO7816-5

  Mainly used at beginning of "communication" between applets

- **public class Util**

Functions for **byte** array manipulation and comparaison

- **arrayCompare()** compares 2 arrays *(in constant time)*
- **arrayCopy()** copies 1 array into another *atomically*
- **arrayCopyNonAtomic()** copies 1 array into another *non-atomically*
- **arrayFillNonAtomic()** fills an array with a **byte** value *non-atomically*

Functions for type conversion (short/byte)

- **short getShort** ()

- **short makeShort**()

- **setShort** ()

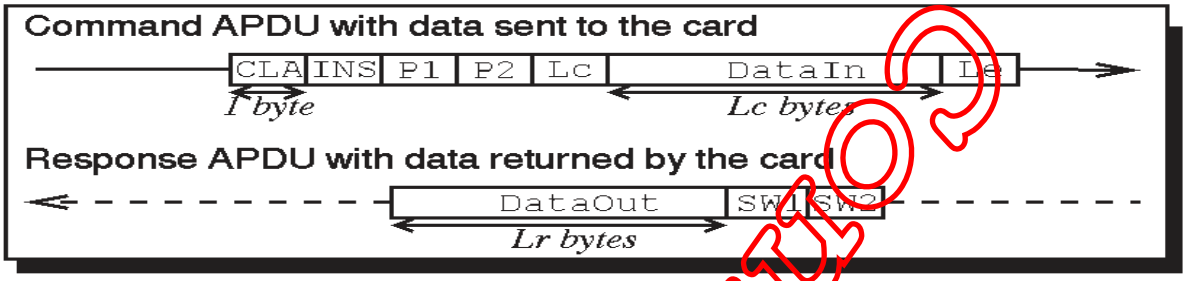
javacard.framework

- **`public final class APDU`**

Encapsulates 7816-4 APDU elements in an I/O buffer

APDUs (Application Programming Data Units) define the format of the «data packets» exchanged between a reader and a card:

- APDUs are defined for both commands and responses to hide the low level details of the protocol used (T=0 ou T=1)
- Status words SW1 et SW2 are standardized (OK = 0x9000)

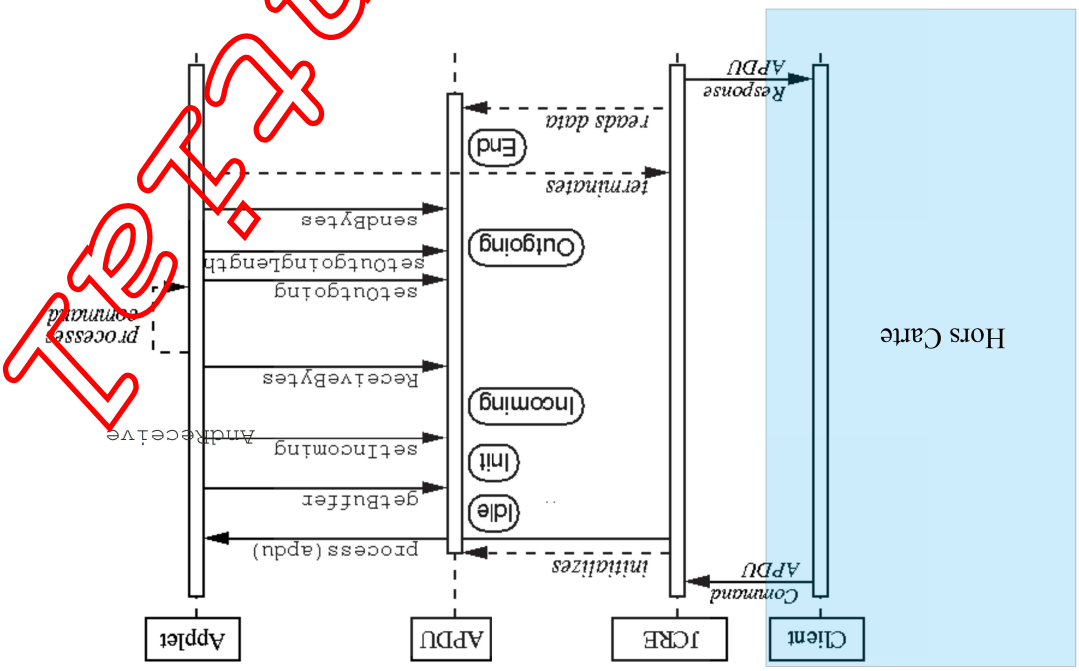


Provides methods for receiving data and sending data

Designed to be protocol independent

The incoming / outgoing APDU data size may be bigger that the APDU buffer size and need to be read/write in portions by the applet

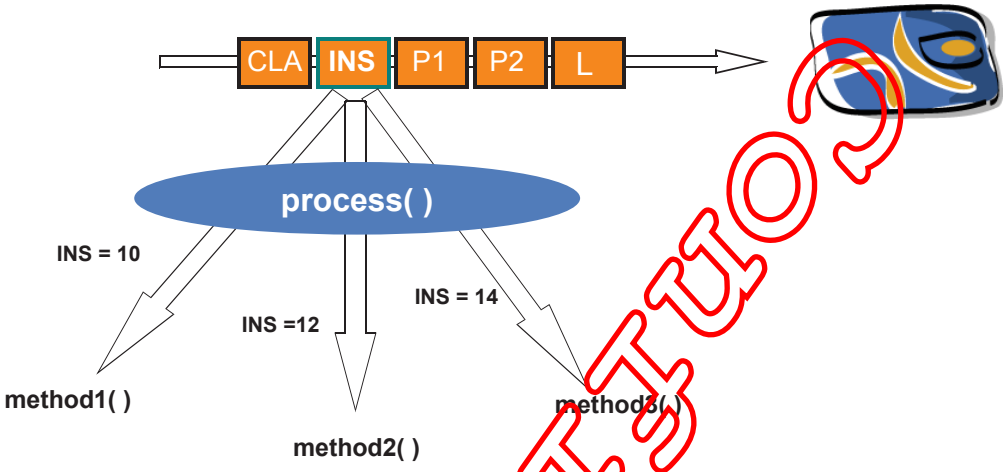
APDU objects are owned by the JCRE (Temporary JCRE entry point objects)



Handling the **APDU** class

Handling the **APDU** class

- `process(apdu APDU)` method is called
- `apdu.getBuffer()`
- `CLA` is checked and then `INS` is examined



`javacard.framework`

- **`public interface ISO7816`**
  Contains only static constants

  Public final static field with `SW_` prefixes define status words (standardized status response of ISO7816-4)

  e.g.: `SW_WRONG_P1P2` // Incorrect parameters (P1,P2) = 0x6B00

  Fields with `OFFSET_` prefixes define constants/offset to use as index in the APDU buffer (byte array)

  e.g.: `OFFSET_CLA` // APDU header offset: CLA = 0

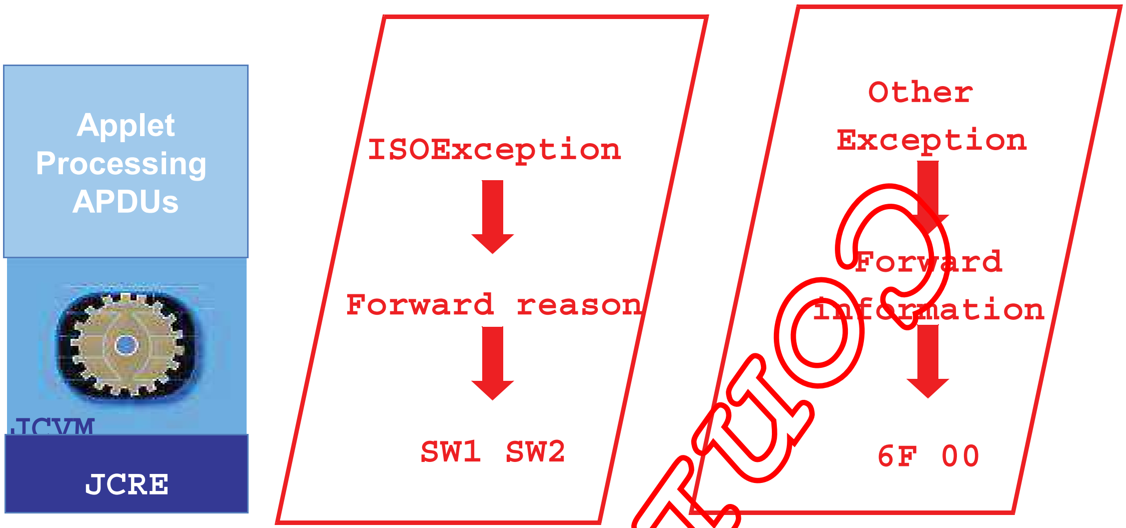
  Fields with "`_00`" suffix require the low order byte to be customized appropriately

  e.g.: `SW_BYTES_REMAINING_00` // Response bytes remaining = 0x6100

- **`public class ISOException`**
  Encapsulates an ISO 7816-4 response status word as its reason code

  `throwIt(short reason)` allows to output a status word error

Java Card Exceptions



javacard.framework

- **`public interface PIN`**
  - Define a PIN behavior and a PIN value
  - Try limit maximum trial of an incorrect PIN before being blocked,
  - Max PIN size, the maximum length of PIN,
  - Try counter, the remaining number of trial,
  - Validated flag => true if a valid PIN has been presented. Flag is reset on every card reset.
  - If a transaction is in progress **update of the try counter shall not participate to the transaction**,

javacard.framework

- **public class OwnerPIN**

Implements interface PIN

Full rights for PIN code management

– Update / Read capabilities -> for the code “owner” only

Features:

– direct modification of the validation flag

– know whether or not the code has been validated

– securely implemented against attacks

Methods available:

- check( )
- isValidated( )
- resetAndUnblock( )
- setValidatedFlag( )
- reset( )
- getTriesRemaining( )
- getValidatedFlag( )
- update( )

javacard.framework

public final class JCSystem

- Static methods (natives) for interaction with the JCRE
- Transactions

Only one level of transaction allowed

beginTransaction(), commitTransaction (), abortTransaction ()

Limited by the RAM and EEPROM capacity

- Transient array

makeTransientXXXArray(lenght, event)

Build a transient array reinitialized at reset or deselection

XXX being `Boolean`, `Byte`, `Short` or `Object`

Event is `CLEAR_ON_DESELECT` or `CLEAR_ON_RESET`

- Object sharing

Build an object that inherit from **shareable**

Provide a reference with **getAppletShareableInterfaceObject (AID, Parameter)**

javacard.framework

- **`public abstract class Applet`**

Super class of all Java Card Applet, user applets must subclass Applet class.

Methods called by the JCRE

User applets MUST implements

- **`static void install(byte[] buf, short off, byte len)`**
  Create a new applet instance
- **`abstract void process(APDU apdu)`**
  Called for requesting the applet isntance to execute a receive command

User applets MAY overrides

- **`boolean select()`**
  Called when the applet instance is selected
- **`void deselect()`**
  Called when the applet instance is selected

Note that:

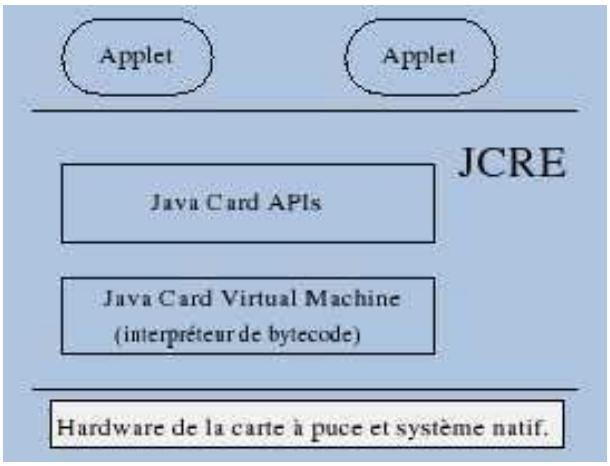
- **`protected void final register()`**
  Called during installation for registering the newly created instance

En résumé !

Problème : contraintes mémoires et faible ressource de calculs

Solutions :
un sous-ensemble des caractéristiques du langage Java
- Au niveau du langage
- Au niveau des APIs

découper la machine virtuelle Java en deux parties
- Une partie interprète embarquée
- Une partie vérification hors carte



Problème : pas de vérificateur embarqué

Solution :
fournir des mécanismes sécuritaires avec l'environnement d'exécution
=> Le firewall !

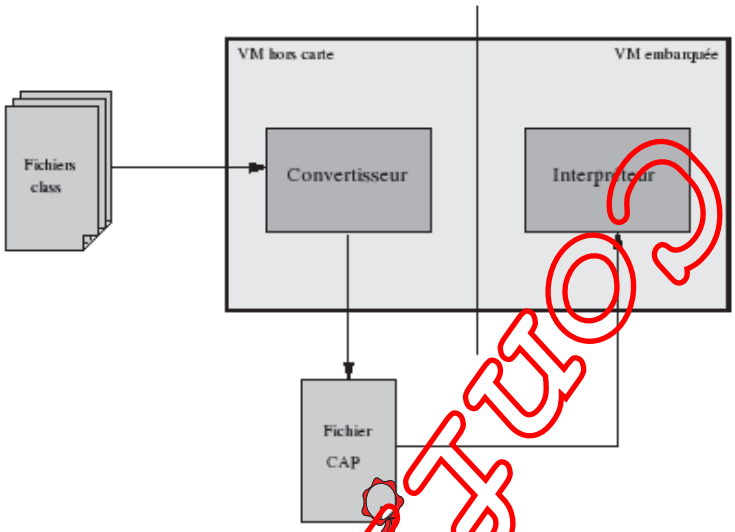
Java Card est un sur-ensemble d'un sous-ensemble du langage Java
Nouveau modèle mémoire utilisant la persistance
APIs spécifiques à la carte

Spécifications Java Card 2.2.1
Virtual Machine
Runtime Environment
API

En résumé !

Définition de la machine virtuelle Java
Vérifieur de Bytecode
Chargeur dynamique de classes
Interpréteur de Bytecode

Les deux parties implémentent toutes les fonctions d'une machine virtuelle.



À cause du découpage de la JCVM, ***la plate-forme est distribuée dans le temps et dans l'espace.***
Il faudra des mécanismes pour garantir que ce qui est produit par la partie hors carte n'est modifiée avant d'arriver à la partie embarquée (mécanisme de DAP – Data Authentification Pattern -- et signature) : cf. GlobalPlatform

La démocratisation

Carte à puce standard

- Application, OS et hardware sont liés
- L'application est développée uniquement par le propriétaire de l'OS
- L'application est développée dans un langage de bas niveau (C, Assembleur)
- Cycle de développement = 5 mois
- Pas de vraie multi-application (données uniquement)

Plate-forme Java Card

- Application, OS et hardware sont indépendants
- L'application est développée par n'importe quel programmeur Java
- L'application est développée dans un langage standard (de haut niveau)
- Cycle de développement = 2 mois
- Carte multi-application (données + code)

Développement Java Card

Construction d'applications carte

Une application carte

Code de la carte (application serveur = applet Java Card)
Code dans le terminal (application cliente)

Construction d'une application Java Card

Construction de l'application serveur (applet)
- Implémentation des services

Installation de l'applet dans les cartes
- Initialisation des services

Construction de l'application cliente
- Invocation des services

Définir le but de l'application puis spécifier le protocole de communication entre l'application cliente et l'applet Java Card.

Commandes à traiter
Réponses possibles
Cas d'erreurs

Construction d'une applet Java Card

Rôle d'une applet

Maintenir son propre état : gestion des champs de l'applet, créer des objets et les référencer pour travailler
Répondre à des APDUs de commande (méthode `process()`) et retourner des APDUs de réponse

Conception

Définir les structures de données nécessaires
Créer les objets de base à l'installation, initialiser les champs
- Implémentation de la méthode `install()`

Définir les APDUs traités par la méthode `process()`
- Implémentation d'un analyseur de commandes
- Utilisation des champs et objets de l'applet
- Utilisation des APIs

Définir les traitements à la sélection et à la desélection

Fabrication (avec un IDE)

Compilation
- Production de fichier class

Conversion
- Production du fichier CAP

Initialisation

Chargement du fichier CAP
Installation de l'applet (instanciation) via un appel à la méthode `install()` en lui donnant un AID

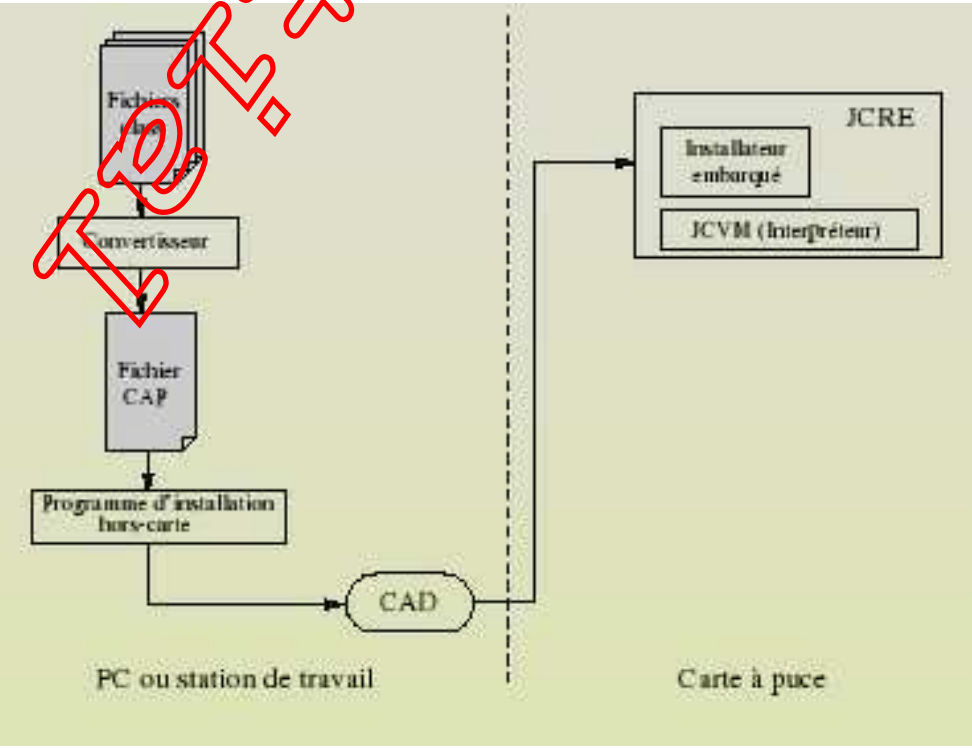
Utilisation

Via l'AID défini pour cette applet

Installateur Java Card

L'installation se fait grâce à deux entités :

l'installateur Java Card et un programme d'installation hors carte



Construction de l'application cliente

Rôle de l'application cliente

Stimuler l'application serveur et utiliser les résultats pour faire son travail (exemple informer le commerçant que oui l'ordre a bien été accepté)

Conception

Définir les structures de données propre à l'application
Choisir son langage, un compilateur, éventuellement un framework développement (.NET, Java, etc)
Choisir un framework de communication : PC/SC, JPC/SC, OCF, javax.smartcardio, etc
Implémentation en utilisant ce framework du protocole de communication avec le serveur (carte)

- Établissement de la liaison avec le serveur : envoie d'un APDU de sélection avec l'AID de l'applet
- Invocation des services de l'applet :
  - Codage et envoie d'APDUs de commande conformes à ceux traités par l'applet
  - Réception et décodage des APDUs de réponse retournés par l'applet

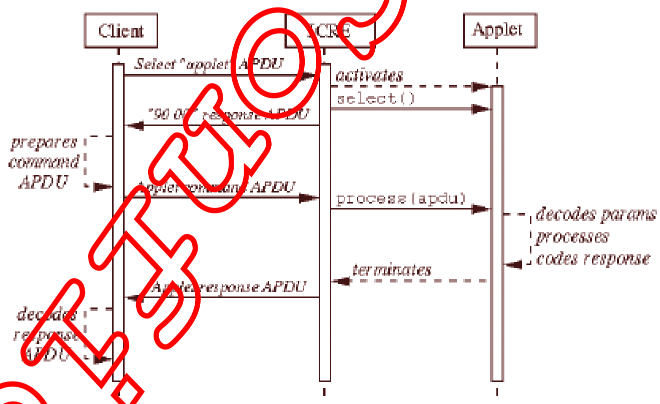
Fabrication

Avec l'IDE choisi

Déploiement

sur les machines où le service sera utilisé

Utilisation



Résumé construction d'applications

Développement de l'applet carte

Utilisation de l'API Java Card
- Essentiellement pour traiter des APDUs

Conversion et installation dans la carte
- Format du bytecode et procédure d'installation dans la carte non «encore» définis par Java Card

Exécution par le JCRE
- Sélection d'applet définie par Java Card
- Réception et transmission d'APDUs par l'applet définies par Java Card

Développement de l'application cliente

Utilisation d'une API de communication carte permettant de transporter des APDUs
- Exemple : API OCF pour Java standard

Sélection de l'applet par son AID
- Commande définie par Java Card

Échanges d'APDUs avec l'applet carte
- Codage d'APDUs de commande et décodage d'APDUs de réponse conformes aux APDUs traités par l'applet

Problèmes pour la construction

Client et applet communiquent par APDUs

Structure de données pauvre (tableau d'octets), peu explicite (pas de typage), et difficile à manipuler
Oblige à définir le contenu et la sémantique des messages échangés : décodage et encodage d'APDUs par le client et l'applet

La spécification des échanges entre client et applet correspond à un protocole

Le format des messages APDUs échangés est utilisé comme spécification commune
Travail sur un protocole plutôt que sur des fonctionnalités
Nécessite une formation carte

L'applet et le client implémentent un protocole

Code «dupliqué» dans les programmes clients et applets
- Dans la carte : toutes les applets décodent des APDUs

Code «sensible»
- Difficile à programmer : prévoir tous les cas, pas d'outils
- Source d'erreurs

Exemple «Compteur» : Conception

État de l'applet :

Maintient une valeur entière positive ou nulle (pas d'objets)

Installation : création de l'objet Applet

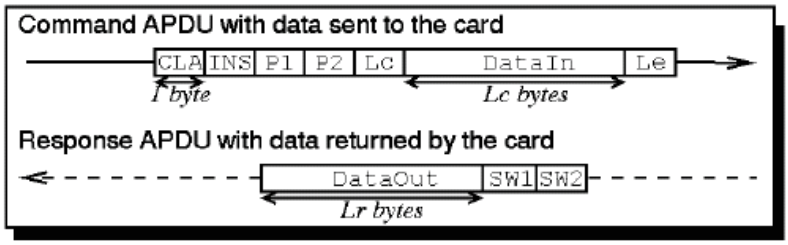
Initialisation de la valeur du compteur
Enregistrement auprès du JCRE

Opérations :

Lecture : retourne la valeur du compteur
Incrémentation/décrémentation : ajoute/soustrait un montant au compteur et retourne la nouvelle valeur du compteur

Sélection et désélection :

Aucun traitement



Rappels :

APDUs traités par l'applet :

int lire()
- Commande : `A0 10 XX XX 00 04`
- Réponse : `RV3 RV2 RV1 RV0 90 00`

int incrementer(int)
- Commande : `A0 12 XX XX 04 AM3 AM2 AM1 AM0 04`
- Réponse : `RV3 RV2 RV1 RV0 90 00`

int decrementer(int) : idem mais `INS=14`

Applet «Compteur» : Classe Applet

```
package fr.cryptis.compteur ;
import javacard.framework.* ;

public class Compteur extends Applet {

    private int valeur;

    public Compteur() { valeur = 0; register(); }

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

    public void process( APDU apdu ) {
        byte[] buffer = apdu.getBuffer();
        if ( buffer[ISO7816.OFFSET_CLA] != (byte)0xA0 )
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
        switch ( buffer[ISO7816.OFFSET_INS] ) {
                case (byte)0x10: ... // Opération de lecture
                case (byte)0x12: ... // Opération d'incrémentation
                case (byte)0x14: ... // Opération de décrémentation
            default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
}
```


Applet «Compteur» : décrémentation

```
case 0x14: // Opération de décrémentation
{
    // Réception des données
    byte octetsLus = apdu.setIncomingAndReceive();
    if ( octetsLus != 4 )
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    int montant = (buffer[ISO7816.OFFSET_CDATA]<<24) |
        (buffer[ISO7816.OFFSET_CDATA+1]<<16) |
        (buffer[ISO7816.OFFSET_CDATA+2]<<8) |
        buffer[ISO7816.OFFSET_CDATA+3];

    // Traitement
    if ( montant<0 || valeur-montant<0 )
        ISOException.throwIt((short)0x6910);
    valeur = valeur - montant;

    // Envoie de la réponse
    buffer[0] = (byte)(valeur>>24);
    buffer[1] = (byte)(valeur>>16);
    buffer[2] = (byte)(valeur>>8);
    buffer[3] = (byte)(valeur);
    apdu.setOutgoingAndSend((short)0, (short)4);
    return;
}
```

Rappel : Applet Java Card

Une applet carte est un programme serveur de la Java Card
APDU de sélection depuis le terminal
Sélection par AID (chaque applet doit avoir un AID unique)

Une fois installée dans la carte, est toujours disponible
Classe qui hérite de `javacard.framework.Applet`
Doit implémenter les méthodes qui interagissent avec le JCRE :
`install()`, `select()`, `deselect()` et `process()`

`public static void install( byte[] bArray, short bOffset, byte bLength )`
Appelée (une fois) par le JCRE quand l'applet est chargée dans la carte
Doit s'enregistrer auprès du JCRE (méthode `register()`)

`public boolean select()`
Appelée par le JCRE quand un APDU de sélection est reçu et désigne cette applet
Rend l'applet active

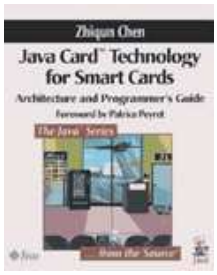
`public abstract void process( APDU apdu )`
Appelée par le JCRE quand un APDU de commande est reçu pour cette applet (doit être active)

`public void deselect()`
Appelée par le JCRE pour désélectionner l'applet courante

Références

Site de la technologie Java Card chez Sun microsystems :
http://java.sun.com/javacard/

Technology for Smart Cards: Architecture and Programmer's Guide
Zhiqun Chen
18 Septembre 2000



Thèse de Damien Sauveron
http://damien.sauveron.fr/

Cours du CNAM (Pierre Paradinas) :
http://deptinfo.cnam.fr/~paradinas/cours/

Cours de Didier Donsez :
http://www-adele.imag.fr/~donsez/cours/#smartcard

Disclaimer

Ce cours a été réalisé grâce :

aux transparents que j'ai pu présenter dans différents séminaires

aux transparents de cours de Samia Bouzefrane, Didier Donsez, Gilles Grimaud, Sylvain Lecomte, Pierre Paradinas et Jean-Jacques Vandewalle.

à quelques illustrations trouvées sur Internet.