JavaCard cryptography

General hints:

- **Use existing algorithms/modes rather than write your own** - Algorithms in JavaCard are much slower and most probably less secure against power analysis than the native functions provided by JavaCard library.
- **Store session data in RAM –** operation in RAM are much faster and more secure against power analysis. Moreover, EEPROM has limited number of rewrites before becomes unreliable (10^5 - 10^6 writes).
- **Do NOT store keys and PINs in primitive arrays –** Use specialized objects like OwnerPIN and Key for storage. These are better protected against power ad fault attacks.
- **Erase unused keys and arrays with sensitive values –** Use specialized method if exists (Key::clearKey()) or overwrite with random data.
- **Use transactions to ensure atomic operations –** Short parts of code that must be executed together should be protected by the transaction. Otherwise, power supply can be interrupted inside code and inconsistency may occur. Be aware of attacks based on interrupted transactions so called Rollback attack.
- **Do not use conditional jumps with sensitive data** – Branching after condition is recognizable with power analysis. E.g., branch THEN increase offset for next instruction only by 1, but branch ELSE must compute new offset dependent on length of THEN code. This addition takes much longer time and is recognizable using power analysis.
- **Allocate all necessary resources in constructor –** Applet installation is usually performed in trusted environment. Will prevent attacks based on limiting resources necessary for applet and thus introducing inconsistency into applet execution.

JavaCard applet for PIN verification

The sample applet implements the following logical steps:

- Allocation of PIN object (OwnerPIN())
- Initial setting of the secret value of PIN (OwnerPIN.update())
- Verification of the correctness of the supplied PIN (OwnerPIN.check())
- Get remaining tries of PIN verification attempts (OwnerPIN.getTriesRemaining())
- Set tries counter to maximum value and unblock blocked PIN. (OwnerPIN.resetAndUnblock())

```
// CREATE PIN OBJECT (try limit == 5, max. PIN length == 4)
OwnerPIN m_pin = new OwnerPIN((byte) 5, (byte) 4);
// SET CORRECT PIN VALUE
m_pin.update(INIT_PIN, (short) 0, (byte) INIT_PIN.length);
// VERIFY CORRECTNESS OF SUPPLIED PIN
boolean correct = m_pin.check(array_with_pin, (short) 0, (byte) array_with_pin.length);
// GET REMAING PIN TRIES
byte j = m_pin.getTriesRemaining();
// RESET PIN RETRY COUNTER AND UNBLOCK IF BLOCKED
m_pin.resetAndUnblock();
```

JavaCard applet for encryption of the supplied data

The sample applet implements the following logical steps:

- Allocation and initialization of the key object (KeyBuilder.buildKey())
- Set key value (DESKey.setKey())
- Allocation and initialization of the object of cipher (Cipher. getInstance(), Cipher. init())
- Receive incoming data (APDU.setIncomingAndReceive())
- Encrypt or decrypt data (Cipher.update(), Cipher.doFinal())
- Send outgoing data (APDU. setOutgoingAndSend())

```
// …. INICIALIZATION SOMEWHERE (IN CONSTRUCT)
// CREATE DES KEY OBJECT
DESKey m_desKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES,
false);
// SET KEY VALUE
m_desKey.setKey(array, (short) 0);

// CREATE OBJECTS FOR ECB CIPHERING
m_encryptCipher = Cipher.getInstance(Cipher.ALG_DES_ECB_NOPAD, false);
// INIT CIPHER WITH KEY FOR ENCRYPT DIRECTION
m_encryptCipher.init(m_desKey, Cipher.MODE_ENCRYPT);
//….

// ENCRYPT INCOMING BUFFER
void Encrypt(APDU apdu) {
    byte[]   apdubuf = apdu.getBuffer();
    short    dataLen = apdu.setIncomingAndReceive();

    // CHECK EXPECTED LENGTH (MULTIPLY OF 64 bites)
    if ((dataLen % 8) != 0) ISOException.throwIt(SW_CIPHER_DATA_LENGTH_BAD);

    // ENCRYPT INCOMING BUFFER
    m_encryptCipher.doFinal(apdubuf, ISO7816.OFFSET_CDATA, dataLen, m_ramArray, (short) 0);

    // COPY ENCRYPTED DATA INTO OUTGOING BUFFER
    Util.arrayCopyNonAtomic(m_ramArray, (short) 0, apdubuf, ISO7816.OFFSET_CDATA, dataLen);

    // SEND OUTGOING BUFFER
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, dataLen);
}
```

JavaCard applet for hashing of the supplied data

JavaCard 2.2.2 standard describes hashing functions MD5, SHA-1, SHA-256 and RIPEMD160. Not all must be implemented by a particular smart card.

The sample applet implements the following logical steps:

- Allocation of the hashing object (MessageDigest.getInstance())
- Reset internal state of hash object (MessageDigest. reset ())

- Update intermediate hash value using incoming array (MessageDigest.update())
- Finalize and read hash value of data (MessageDigest.doFinal())

```
// CREATE SHA-1 OBJECT
MessageDigest m_sha1 = MessageDigest.getInstance(MessageDigest.ALG_SHA, false);

// RESET HASH ENGINE
m_sha1.reset();
// PROCESS ALL PARTS OF DATA
while (next_part_to_hash_available) {
 m_sha1.update(array_to_hash, (short) 0, (short) array_to_hash.length);
}
// FINALIZE HASH VALUE (WHEN LAST PART OF DATA IS AVAILABLE)
// AND OBTAIN RESULTING HASH VALUE
m_sha1.doFinal(array_to_hash, (short) 0, (short) array_to_hash.length, out_hash_array, (short) 0);
```

JavaCard applet for computation of MAC based on symmetric cryptography

Various cryptographic checksum algorithms are implemented by the card (see javacard.security.Signature).

The sample applet implements the following logical steps:

- Allocation of the signature object (Signature.getInstance ())
- Allocation of the key object used for signature (KeyBuilder.buildKey ())
- Set key for usage with signature object in SIGN mode (Signature.init ())
- Generation of MAC over buffer (Signature.sign())

```
private   Signature      m_sessionCBCMAC = null;
private   DESKey         m_session3DesKey = null;

// CREATE SIGNATURE OBJECT
m_sessionCBCMAC = Signature.getInstance(Signature.ALG_DES_MAC4_NOPAD, false);
// CREATE KEY USED IN MAC
m_session3DesKey = (DESKey) KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
KeyBuilder.LENGTH_DES3_3KEY, false);

// INITIALIZE SIGNATURE DES KEY
m_session3DesKey.setKey(m_ram, (short) 0);
// SET KEY INTO SIGNATURE OBJECT
m_sessionCBCMAC.init(m_session3DesKey, Signature.MODE_SIGN);

// GENERATE SIGNATURE OF buff ARRAY, STORE INTO m_ram ARRAY
m_sessionCBCMAC.sign(buff, ISO7816.OFFSET_CDATA, length, m_ram, (short) 0);
```

JavaCard applet for signing of the supplied data

JavaCard 2.2.2 standard describes various combination of signature functions based on asymmetric cryptography (RSA, DSA, ECDSA) and symmetric cryptography (MAC – DES, AES based). Again, not all must be implemented by a particular smart card.

The sample applet implements the following logical steps:

- Allocation of the key and signature objects (KeyBuilder.buildKey, new KeyPair, Signature.getInstance)
- On-card generation of key pair (KeyPair.genKeyPair())
- Obtaining references to private and public key (KeyPair.getPrivate/Public)
- Initialization of signature engine with private key (Signature.init)
- Performing signature operation (Signature.update, Signature.sign)

```
// CREATE RSA KEYS AND PAIR
m_privateKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,KeyBuilder.LENGTH_RSA_1024,false);
m_publicKey = KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PUBLIC,KeyBuilder.LENGTH_RSA_1024,true);
m_keyPair = new KeyPair(KeyPair.ALG_RSA, (short) m_publicKey.getSize());

// STARTS ON-CARD KEY GENERATION PROCESS
m_keyPair.genKeyPair();
// OBTAIN KEY REFERENCES
m_publicKey = m_keyPair.getPublic();
m_privateKey = m_keyPair.getPrivate();

// CREATE SIGNATURE OBJECT
Signature m_sign = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
// INIT WITH PRIVATE KEY
m_sign.init(m_privateKey, Signature.MODE_SIGN);

// SIGN INCOMING BUFFER
signLen = m_sign.sign(apdubuf, ISO7816.OFFSET_CDATA, (byte) dataLen, m_ramArray, (byte) 0);
```

JavaCard applet for generating random data

JavaCard 2.2.2 standard defines two types of random generators within object RandomData: RandomData.ALG_SECURE_RANDOM and RandomData.ALG_PSEUDO_RANDOM. Sometimes, the ALG_PSEUDO_RANDOM is not implementated by the card.

The sample applet implements the following logical steps:

- Allocation of the random data object (RandomData.getInstance())
- Generation of random block with given length (RandomData.generateData())

```
private RandomData     m_rngRandom = null;

// CREATE RNG OBJECT
m_rngRandom = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);

// GENERATE RANDOM BLOCK WITH 16 BYTES
m_rngRandom.generateData(m_testArray1, (short) 0, ARRAY_ONE_BLOCK_16B);
```

E-Commerce & E-Payment Security

JAVA CARD TECHNOLOGY & EMV

cristian.toma@ie.ase.ro – Business Card

Cristian Toma

IT&C Security Master

Dorobantilor Ave., No. 15-17
010572 Bucharest - Romania
http://ism.ase.ro
cristian.toma@ie.ase.ro
T +40 21 319 19 00 - 310
F +40 21 319 19 00

Java

Sun

ORACLE

IT&C SECURITY MASTER

ISM

ism.ase.ro

Scientia potestas est

Organization Form

Didactic Activities: Lectures 50% + Laboratory 50%

Teaching Language: English

Evaluation Methodology: Written Exam + Project
60% | 40% |

Objectives: Creating the ability to understand the theoretical and practical issues of the design security models and applications for reliable e-commerce and e-payment systems.

E-Commerce and E-Payment Security Agenda

Section I – Java Standard Programming – DONE ALREADY in SEMESTER 1

- Memory model
- Cloning, Inheritance, Polymorphism and late-binding
- Generics
- Minimal Java Collection Framework
- I/O Stream Classes (Native methods – JNI + Java Lib Dev)
- Factory Methods
- Exception Handling

Section II – Smart Cards & Java Card Technology

- Complete Java Card Application
- Smart Card Basics
- Standards, SDK and APIs
- Java Card Applet – Card-let Life Cycle
- Minimal Wallet sample – compilation, conversion, loading and simulation
- Java Card Technology Overview
- Java Card Objects
- Atomicity and Transaction
- Java Card Exceptions and Exception Handling
- Applet Firewall and Object Sharing
- Programming Cryptography
- Java Card Remote Method Invocation - obsolote

Section III – E-Commerce & E-Payment Smart Card Integration

- Models and implementations
- SET & 3D Secure, EMV

References

Part I – Java Standard Programming

- Lecture: Java Secure Programming – S1
- Tutorials from java.sun.com – now OTN

Part II – Smart Cards & Java Card Technology

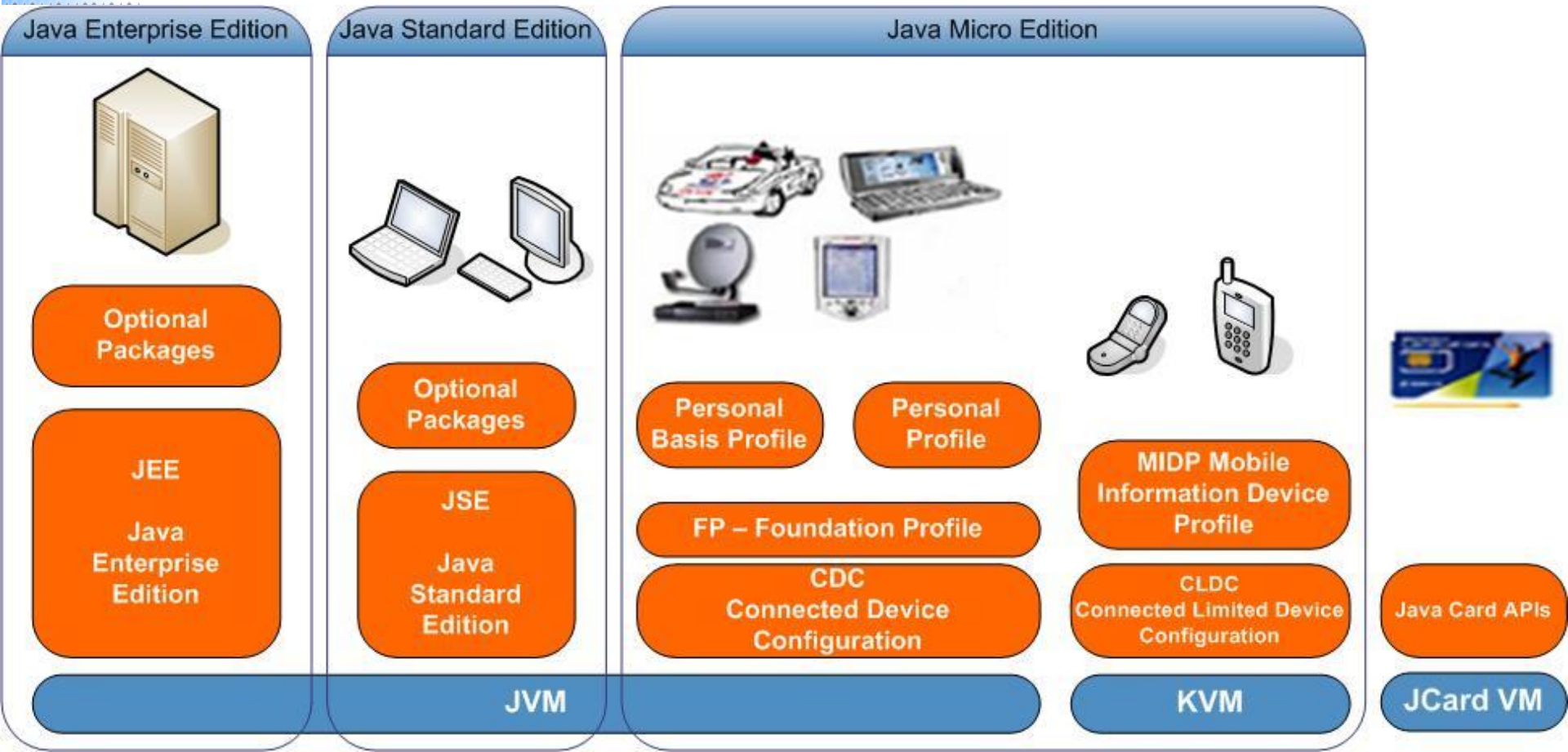
- Zhiqun Chen, "Java Card Technology for Smart Cards", Addison Wesley Publishing House - 2004
- Wolfgang Rankl & Wolfgang Effing, "Smart Card Handbook" – Third Edition, Willy Publishing House - 2003
- Java Card Tutorials from java.sun.com

Part III – E-Commerce and E-Payment Smart Card Integration

- Internet resources

Java Technologies

Java

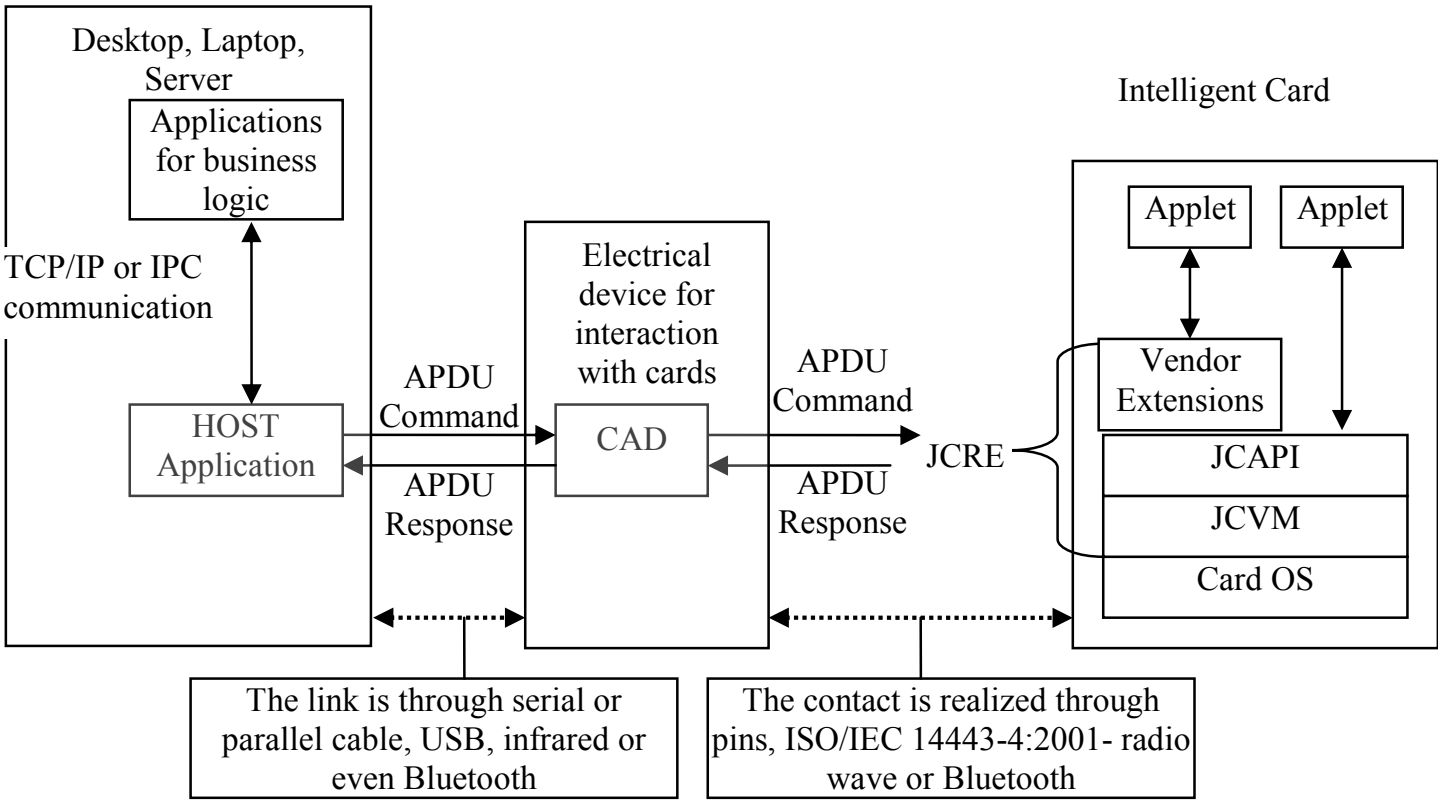


Java Essentials

- Memory model
- Inheritance, Polymorphism and late-binding
- Generics
- Minimal Java Collection Framework
- I/O Stream Classes (Native methods – JNI + Java Lib Dev)
- Factory Methods
- Exception Handling
- Crypto Programming

SEE LECTURE ABOUT JAVA from SEMESTER 1

Complete Java Card Application



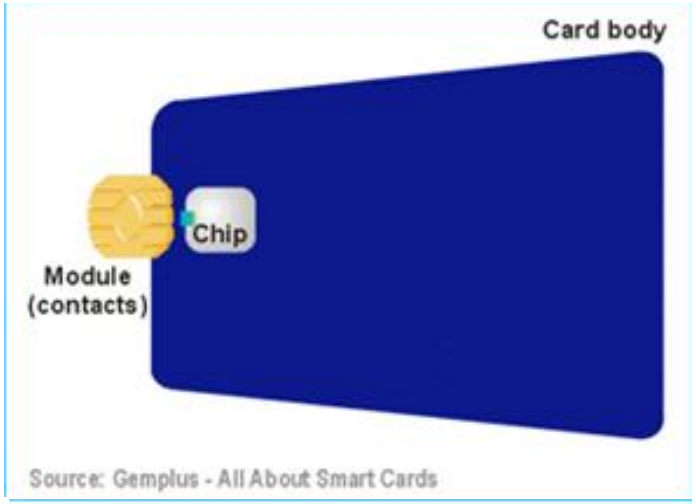
Legend:

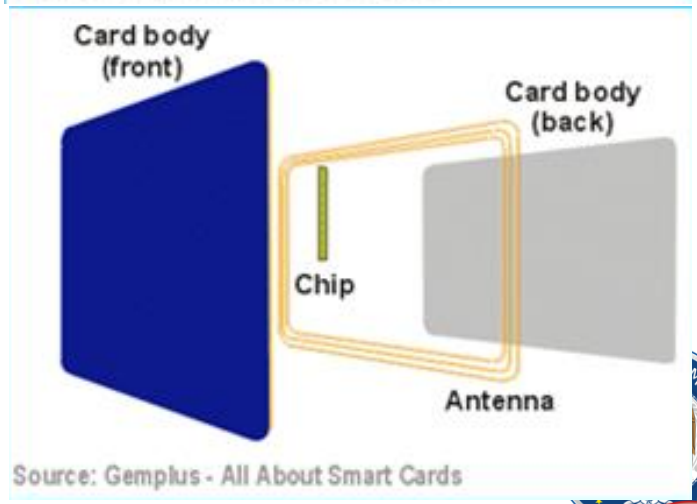
JCVM=Java Card Virtual Machine
JCAPI=Java Card Advanced Programming Interface
JCRE =Java Card Runtime Environment
CAD=Card Acceptance Device
IPC=Inter-Process Communication
APDU=Application Protocol Data Unit

Smart Card Basics

Cards with memory chip, contains different data but can not compute the store data because the card don't have a microprocessor. They are fully depended by the host application.

Cards with microprocessor chip, short term chip cards, contains a microprocessor which is used for computations. Besides this microprocessor with 8, 16 or 32 bits register, the card could contain one or more memory chips which is using for read-only memory and for random access memory – RAM. This features offer to a card almost the power of a desktop computer. This type of cards are used in different informatics systems like banking credit cards, cards for access control in institutions, SIMs – Secure Identification Module – for mobile phones and cards for accessing digital TV networks.

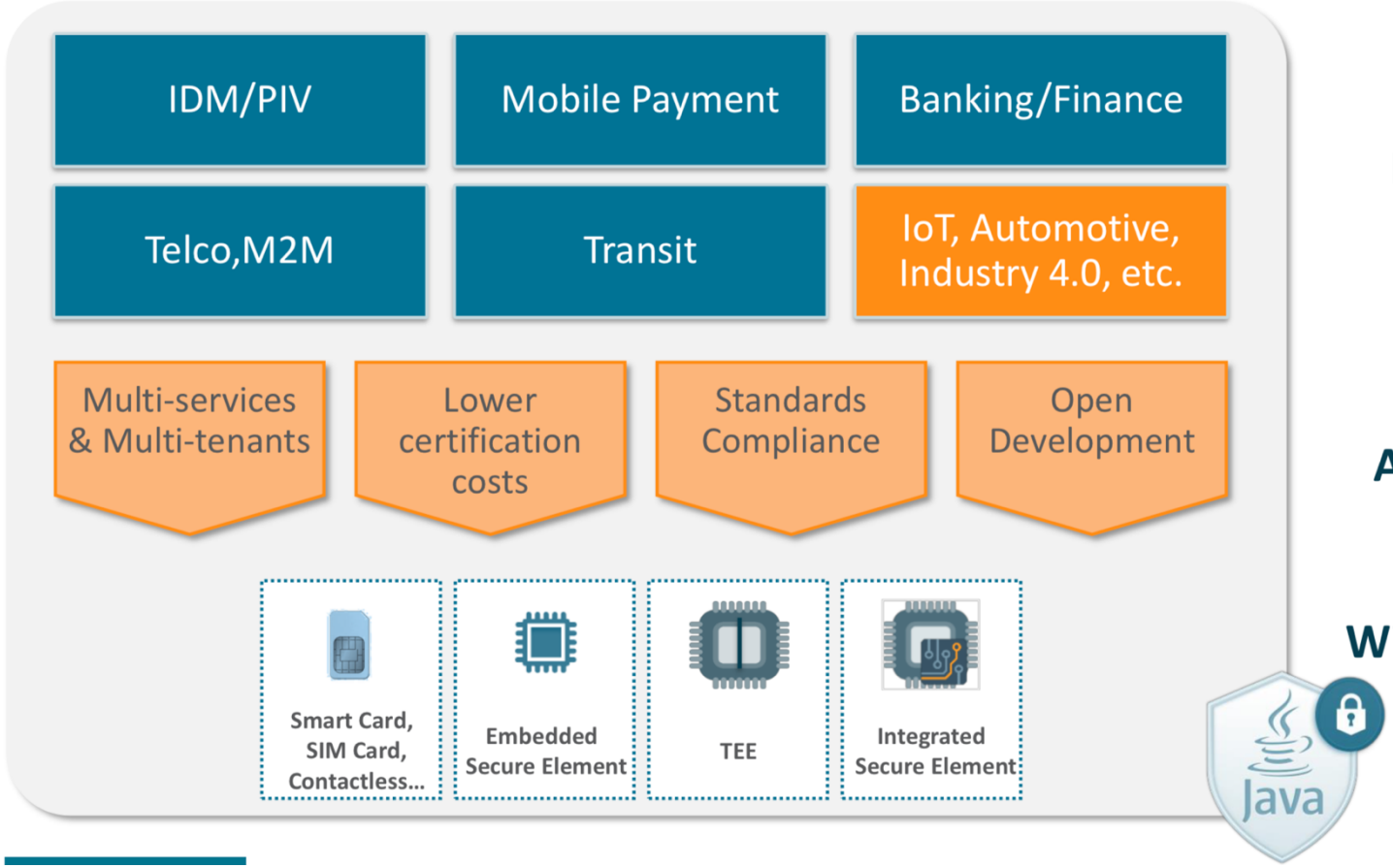




(U)SIM Overview

(U)SIM JavaCard

Enabling Security with the Java Card Platform



In Vertical Markets

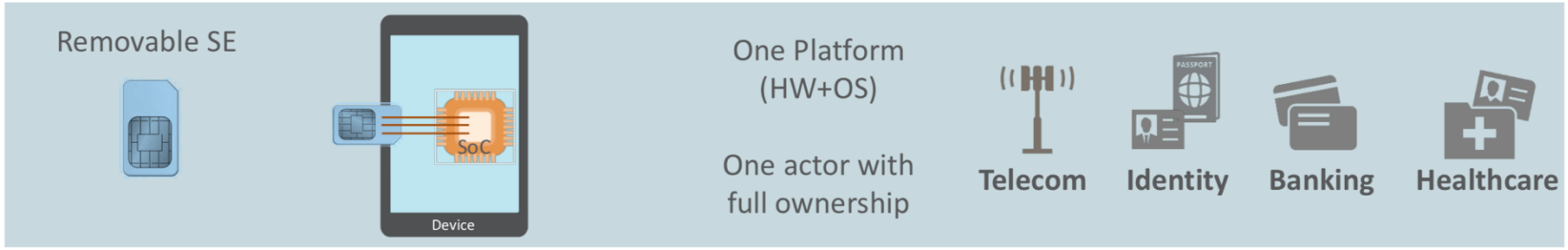
Via Secure Open Application Platform

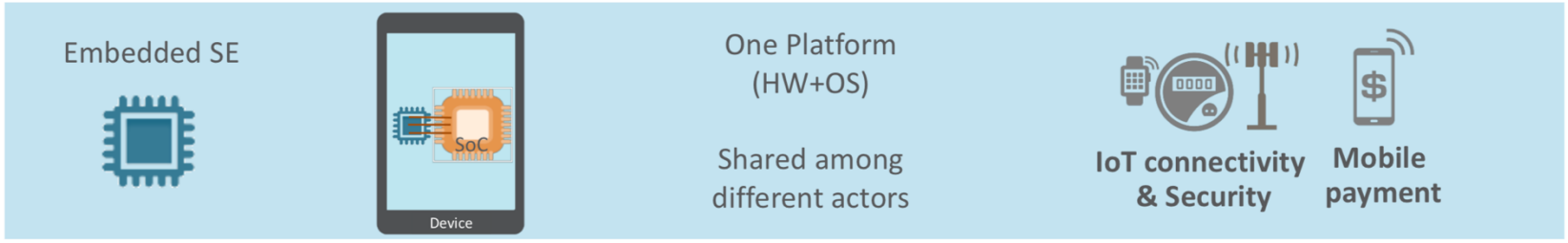
With Choice of Security Form Factors

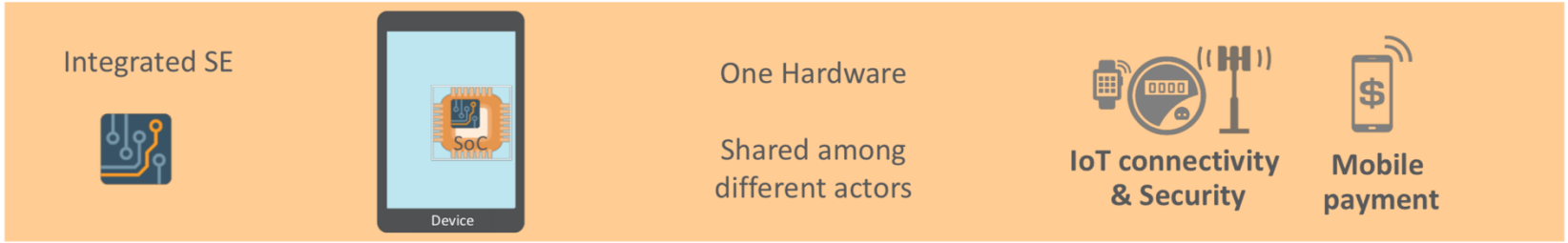
(U)SIM Overview

(U)SIM Secure Element

Secure Element: Form Factor evolutions



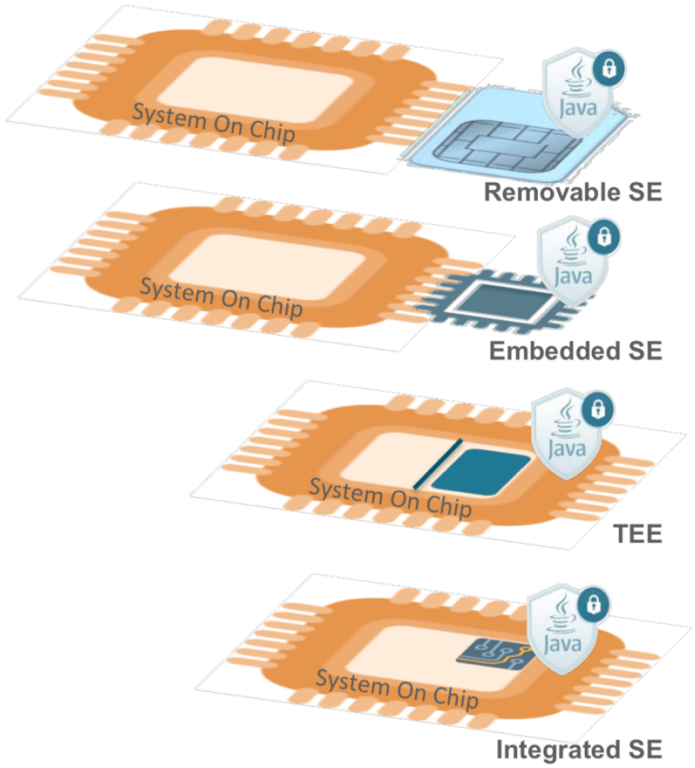




(U)SIM Overview

(U)SIM JavaCard

Java Card as Unified Security Framework

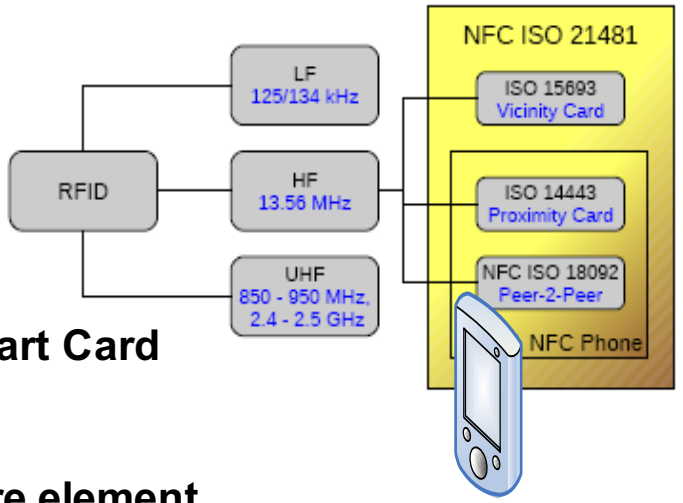


Java Card

- **Scalable Architecture for embedded security**
  - Certifiable platform can be applied across secure segments
  - Small footprint enables any form factor
- **Standards based implementation and certification**
  - Public specifications (Oracle, GP, ETSI) and verifiable compatibility
  - Certified protection profile as standard input for product security targets
- **Proven, extensible and Manageable platform**
  - Wider range of available solutions, tools and expertise
  - Ability to deploy and manage applications from different providers in the value chain (chip maker, OEM, MNO, SSP, user)
- **Content portability across hardware form Factors**
  - Service development / deployment is abstracted from the target HW
  - Easy Migration path for existing applications : eUICC and Payment
  - Hardware choice becomes a factor of commercial and security requirements

Smart Card World

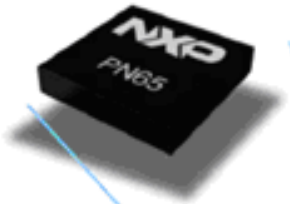
NFC/RFID Mobile Implementation Case

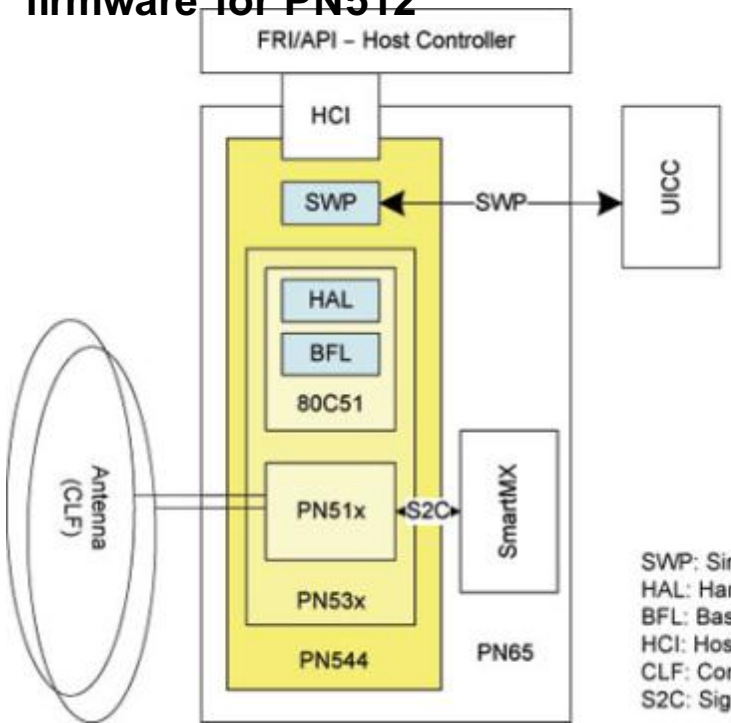


NXP PN65:

- **SmartMX - P5CN072 Secure Dual Interface PKI Smart Card Controller – Java Card**
- **PN 544**
  - **SWP – Single Wired Protocol for SIM as secure element**
  - **PN531**
    - **PN 512 - NFC transmission module @ 13,56MHz**
    - **Microcontroller 80C51 – 32KB ROM and 1KB RAM running firmware for PN512**

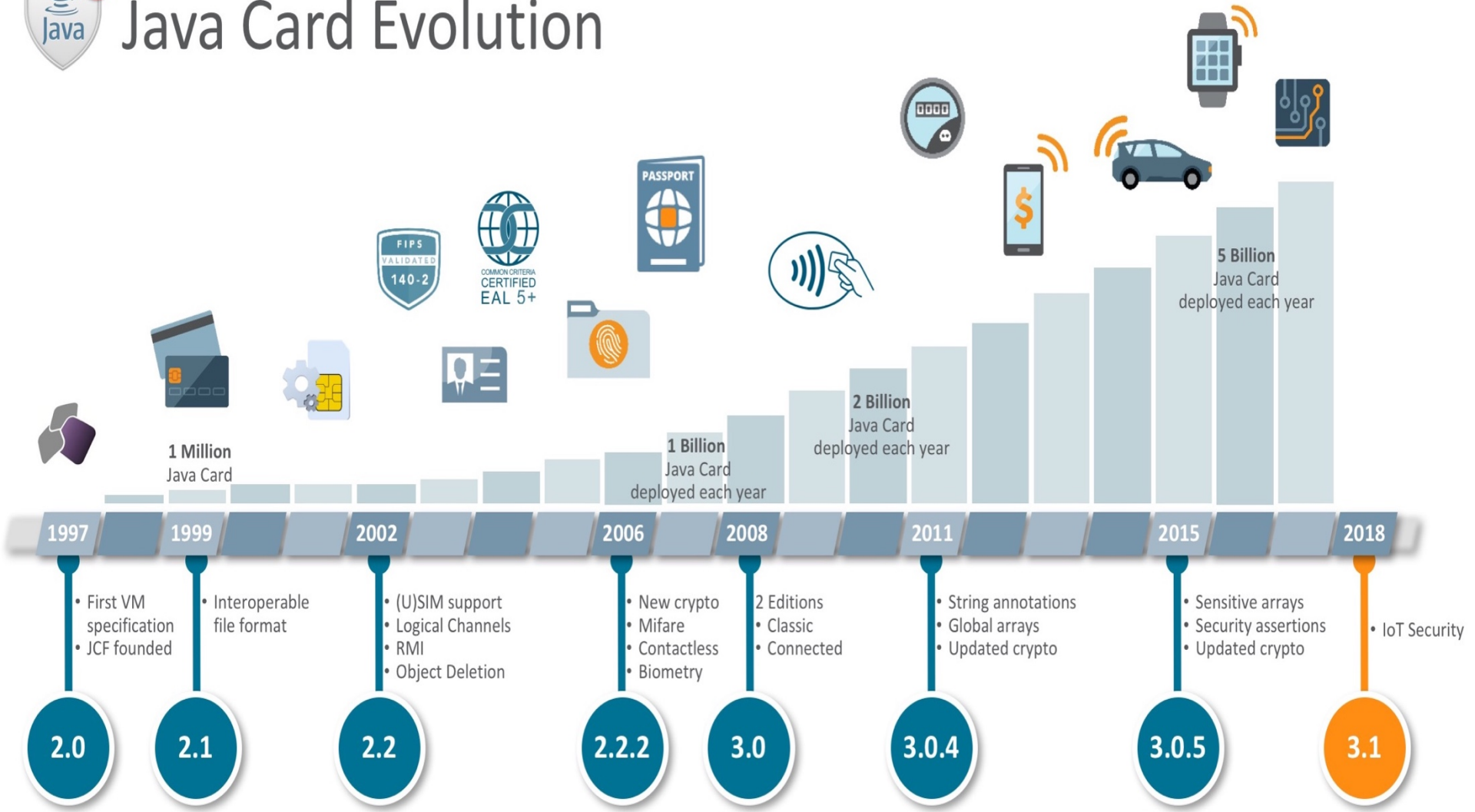
***Samsung Nexus S (Android OS) + ISO 15693 + Proprietary P2P
Nokia C7 (Symbian OS)



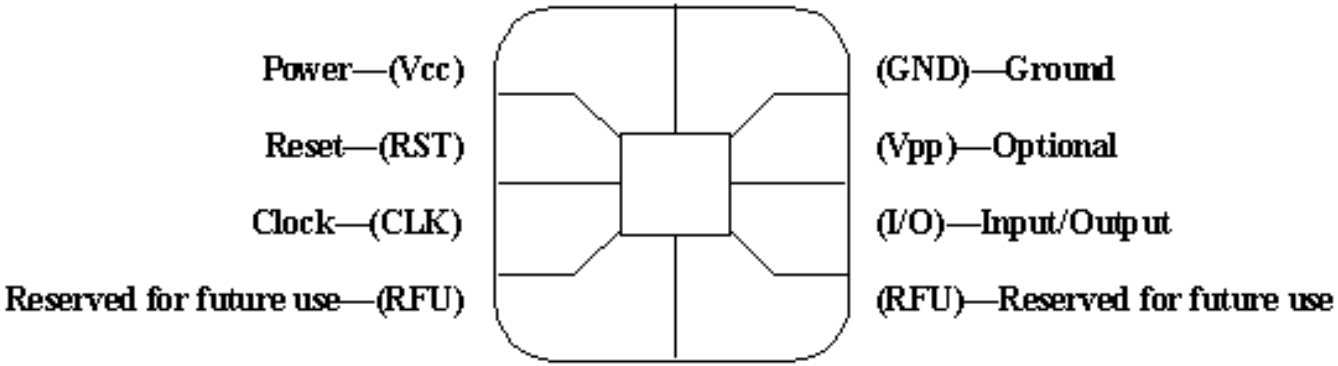


SWP: Single Wire Protocol
HAL: Hardware Abstraction Layer
BFL: Basic Function Library
HCI: Host Controller Interface
CLF: Contactless Frontend
S2C: SigIn-SigOut-Connection

Java Card Evolution



Smart Card Basics – Contact Points



1. **Vcc** – supplies power to the chip. Vcc voltage is 3 or 5 volts.
2. **RST** – is used for sending the signal to reset the microprocessor – this is called **warm reset**. A **cold reset** is done by switching the power supply on and off.
3. **CLK** – smart card microprocessor do not have internal clock generation. The CLK point supplies the external clock signal from which the internal clock is derived.
4. **GND** – is reference voltage, considered 0.
5. **Vpp** – optional point used for older cards to program EEPROM with idle/active state.
6. **I/O –** input / output access point in half duplex mode.

Smart Card Basics – Memory System

1. **ROM** – Read Only Memory – used for storing fixed programs of the card, including the OS Kernel. No electrical power is needed for maintaining data and can not be written after it is manufactured. The process of writing binary image (including data and programs) into the ROM is called ***masking***.

2. **EEPROM** – Electrical Erasable Programmable Read Only Memory – can preserve data when the power supply is off. Can be modified after it is manufactured. Briefly can be considered the "hard-disk" of the card. Accept at least 100000 reliable writing cycles and can retain data for 10 years. Reading the EEPROM is faster than RAM but writing in EEPROM is 1000 time slower than RAM.

3. **RAM** – Random Access Memory – used as temporary space for modifying and storing data mode. RAM is non persistent memory. The content is not preserved when the power is removed from memory cells. Unlimited number of access.

4. **Flash Memory** – used instead of EEPROM.

Java Smart Cards Communication Model

- Message Passing Model
  - APDU Commands
  - APDU Responses
- JCRMI – Java Card Remote Method Invocation
- SATSA – Security and Trust Services API. SATSA defined in JSR 177

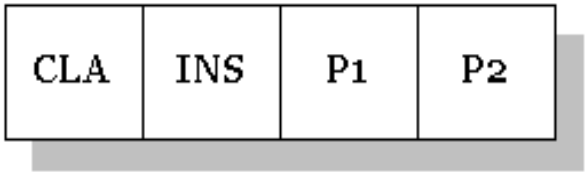
Message Passing Communication Model

APDU Command						
CLA (1 byte)	INS (1 byte)	P1 (1 byte)	P2 (1 byte)	Lc (1 byte)	Data Field (as many as is indicating Lc field)	Le (1 byte)
Header-mandatory				Body-optional		

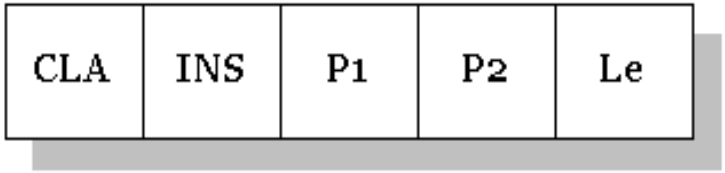
APDU Response		
Data Field (max bytes as specified in Le field from the APDU Command)	SW1 (1byte)	SW2 (1 byte)
Body-optional	Trailer-mandatory	

APDU Commands and Responses Cases

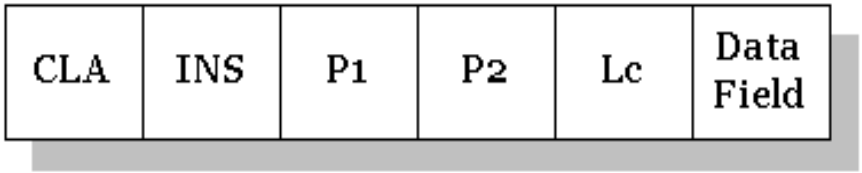
Case 1:
No Command data,
No Response required



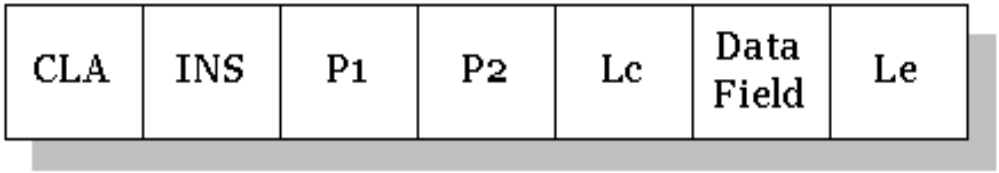
Case 2:
No Command data,
Yes Response required



Case 3:
Yes Command data,
No Response required



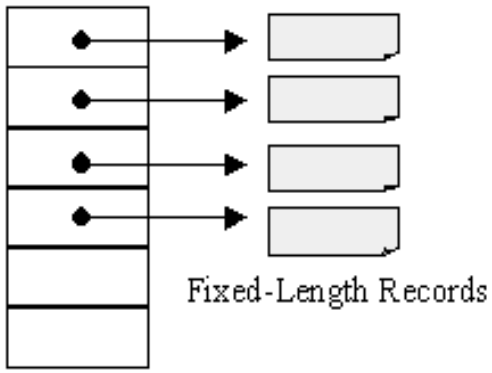
Case 4:
Yes Command data,
Yes Response required



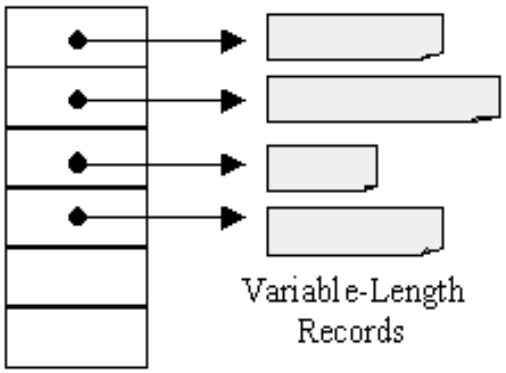
Smart Card File Systems

Class	File Type	Description
DedicatedFile	Dedicated	Act as containers for elementary files.
ElementaryFile	Elementary	Store references to the byte arrays that contain the data.
LinearVariableFile	Linear	Stores variable-length linear records.
LinearFixedFile	Linear	Stores fixed-length linear records.
CyclicFile	Cyclic	Stores fixed-length cyclic records.

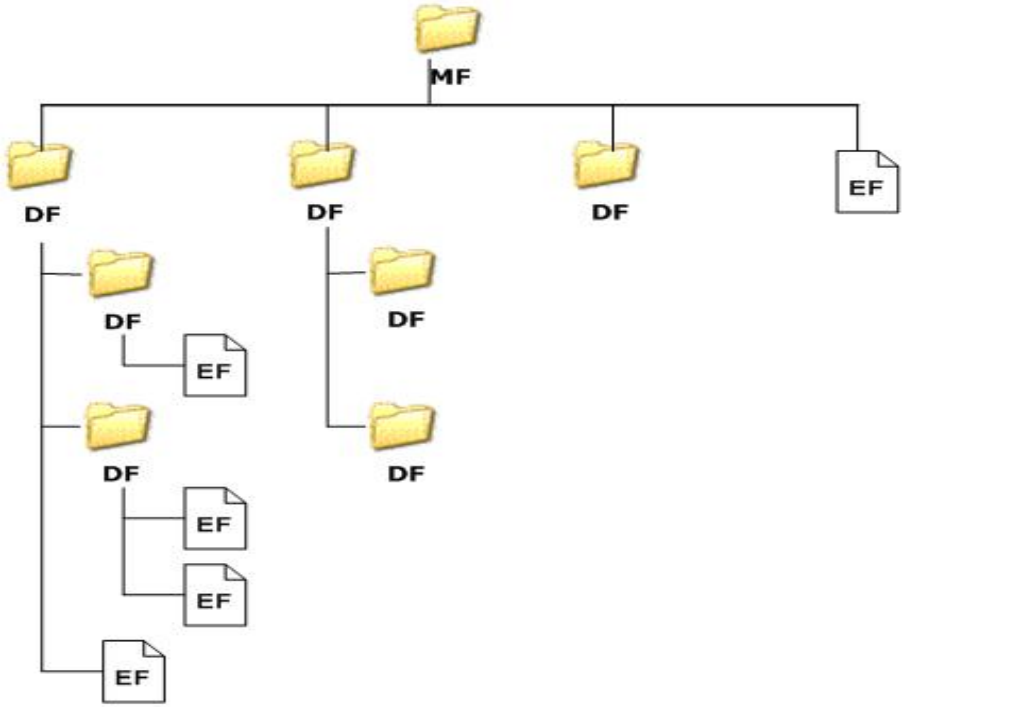
Linear Fixed File



Linear Variable File



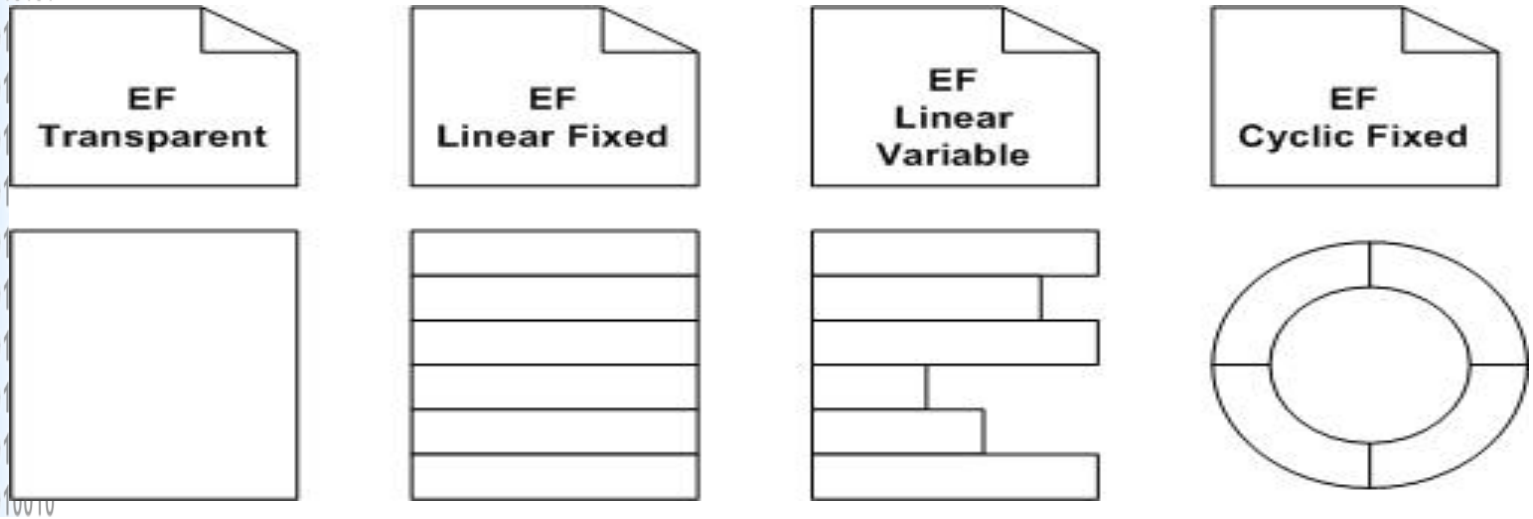
Smart Card File Systems

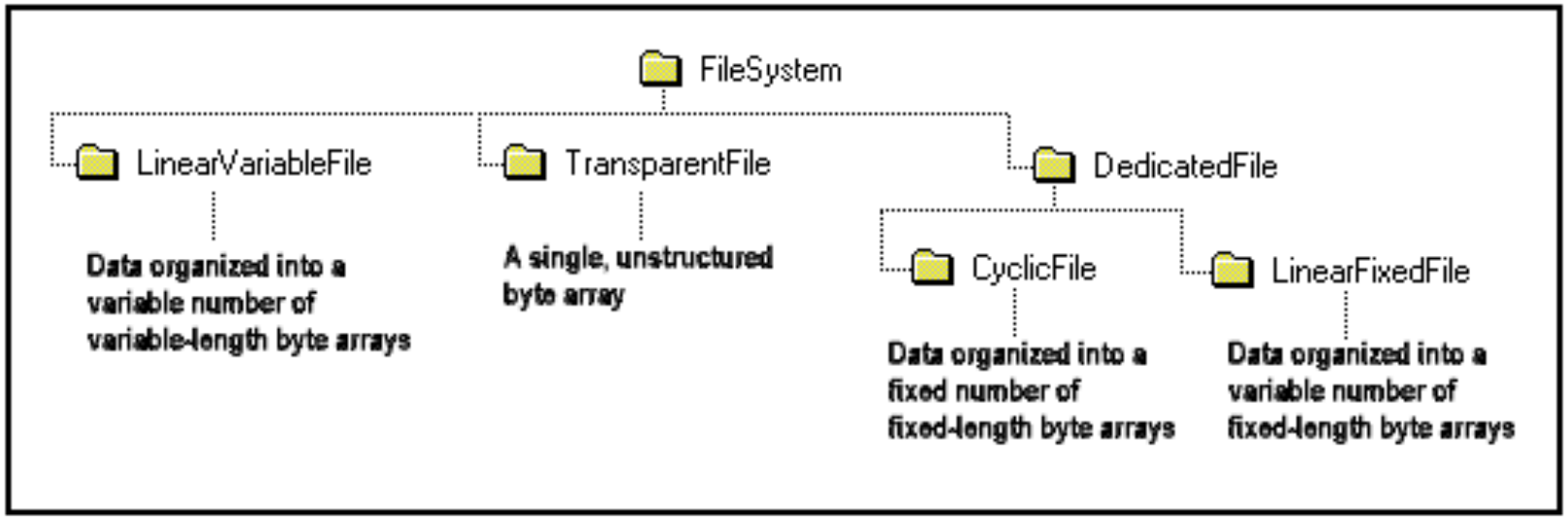


File Names and Identifications according with ISO/IEC 7816-4

MF – Master File	DF – Dedicated File	EF – Elementary File
FID – File Identifier	FID – File Identifier Or DF Name – may include an AID	SFI – Short FID Or FID

Applet Data Hierarchy





Development Artifacts - Standards, SDK and APIs

Product	Company	WWW
JavaCard SDK **Classic vs. Connected**	Sun / Oracle	**http://www.oracle.com/technetwork/java/embedded/javacard/downloads/javacard-sdk-2043229.html**
JCOP Smart Card	IBM -> NXP Philips	tools.jcop@nxp.com
PC/SC **MUSCLE**	Microsoft Linux	www.pcscworkgroup.com **www.pcsclite.alioth.debian.org**
GemAlto GemXplore for SIMs	Gemplus + Axalto	www.gemalto.com **https://webstore.gemalto.com**
Mifare NXP Philips RFID SDK	OMNIKEY Readers + ARYGON	www.omnikey.com
Global Platform	Global Platform Group	www.globalplatform.org https://sourceforge.net/projects/gpj/ https://github.com/martinpaljak/GlobalPlatformPro
MOLTOS SDK	MULTOS	www.multos.com
EMV VISA 4.x	EMV	www.emvco.com

Development Artifacts - Standards, SDK and APIs

OpenJDK 11:
https://download.java.net/openjdk/jdk11/ri/openjdk-11+28_windows-x64_bin.zip

Eclipse 2020-03:
https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2020-03/R/eclipse-java-2020-03-R-win32-x86_64.zip&mirror_id=1156

Java Card 3.1.0 Tools + Simulator:
https://www.oracle.com/java/technologies/javacard-sdk-downloads.html

SHA Online simulator:
http://www.sha1-online.com/
aaaaaaaaaabbbbbbbbbb
=>
c9ba0f7d724228c8b6a410f87135d379da33eb87

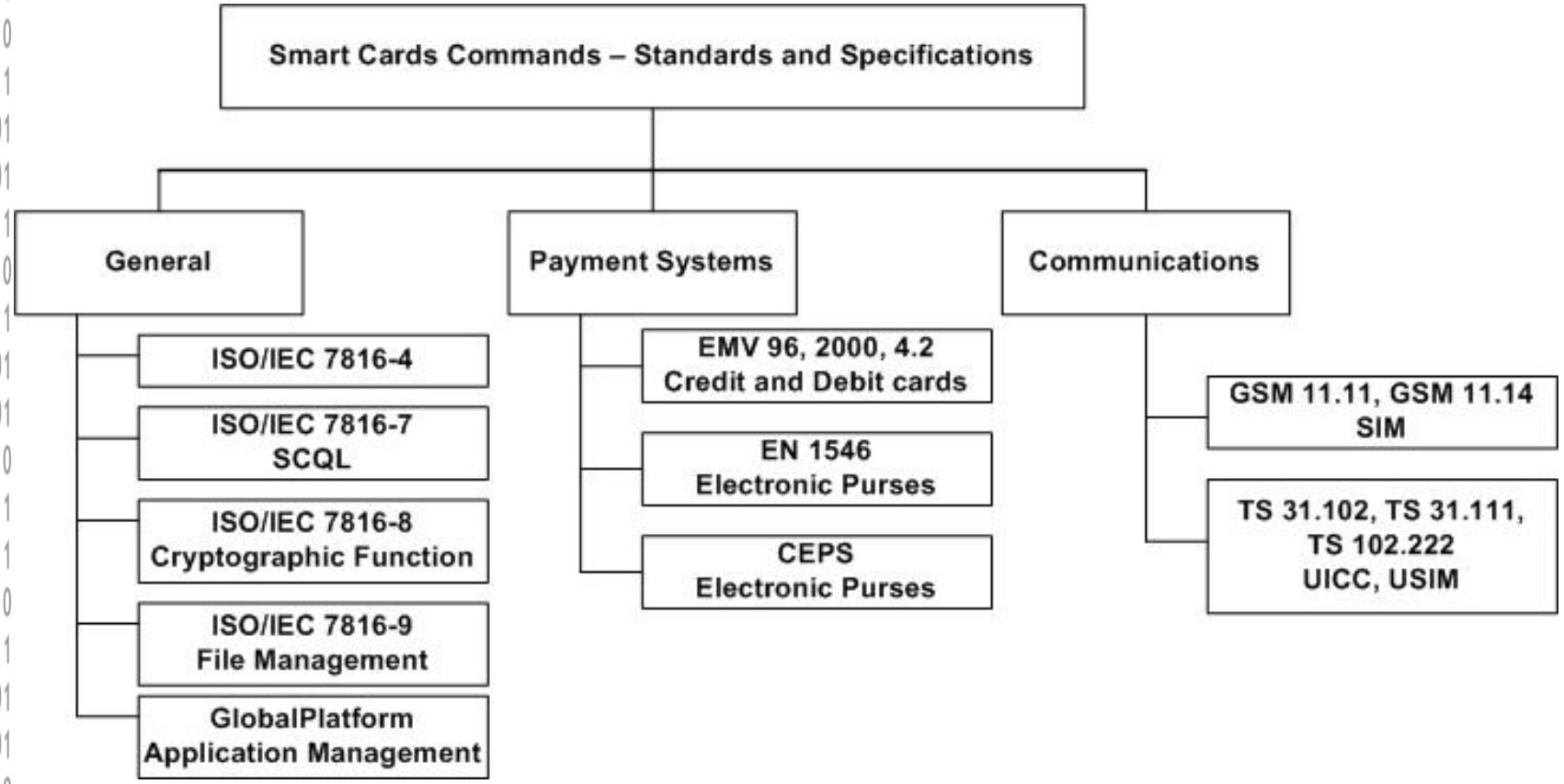
Standards, SDK and APIs

- ISO 7816 – "Identification cards – Integrated Circuits cards with Contacts" – ISO
  - Part 1 – physical layer
  - Part 2 – dimension and location of the contacts
  - Part 3 – electronic signals and transmission protocols
  - Part 4 – inter-industry commands for interchange
  - Part 5 – application identifiers
  - Part 6 – inter-industry data elements
- ISO 14443 – RFID Cards – Mifare & Calypso
- GSM – Global System 4 Mobile Communication – ETSI
  - GSM 11.11 – specification 4 SIM mobile equipment interface
  - GSM 11.14 – specification 4 SIM application toolkit
  - GSM 03.48 – security mechanism 4 SIM application toolkit
  - GSM 03.19 – SIM API 4 Java Card Platform
  - SATSA – JSR 177

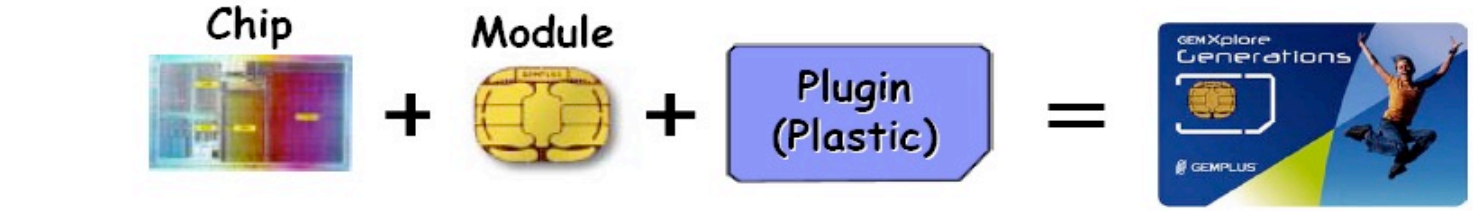
Standards, SDK and APIs

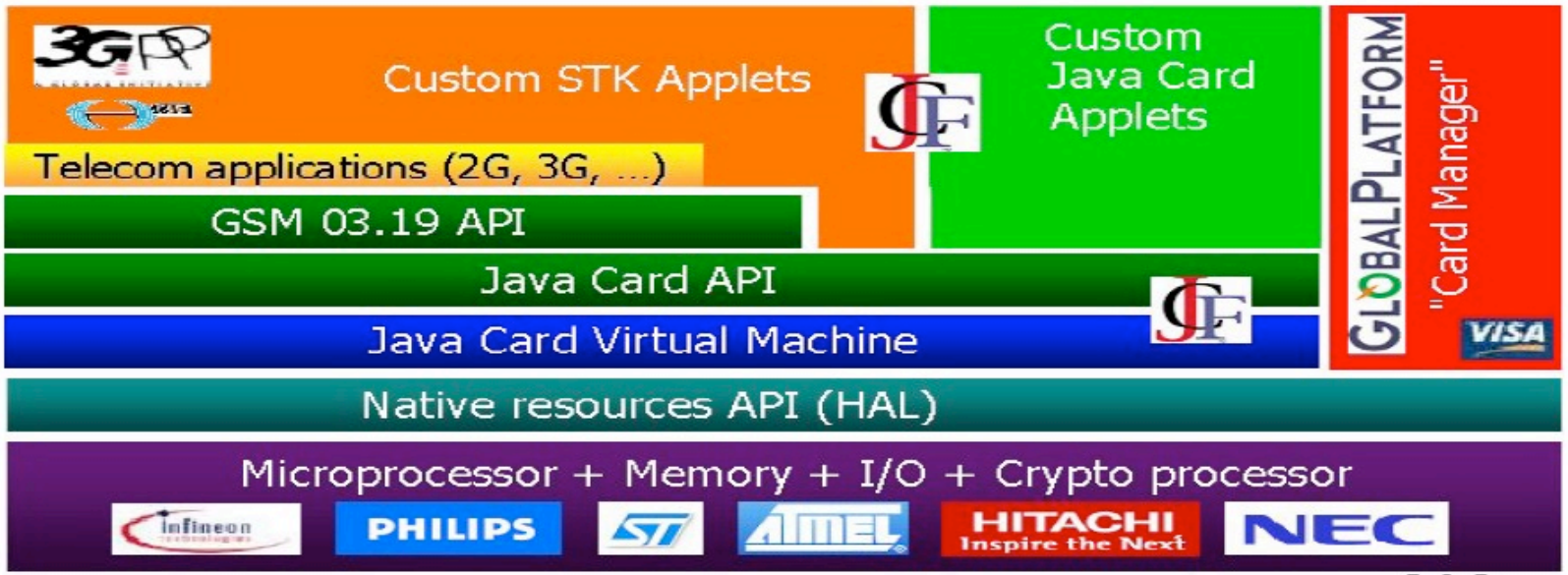
- **EMV – Europay, Mastercard and Visa**
  - EMV '96 – Integrated Circuits Card Specification
  - EMV '96 – Integrated Circuits Card Terminal Specification
  - EMV '96 – Integrated Circuits Card Application Specification 4 Financial Industry
  - EMV 4.1 and 4.2 the last specifications reunited in 4 books
- **Open Platform**
  - Specification transferred to the GlobalPlatform 4 card specification and card terminal specification. Specifically defines the off-card communication with terminal and on-card application management
- **Open Card Framework – JCOP NXP Philips – Host side API**
  - Initially produced by IBM, currently by OpenCard Consortium preovides the standard interface for the interactions between card readers and the application.
- **PC/SC**
  - Developed by the PC/SC Workgroup defines a general purpose architecture for using smart-cards on personal computer system in which the smart card applications are build on top one or more service providers and a resource manager.

Standards, SDK and APIs



SIM Technologies – Best Java Card in Market

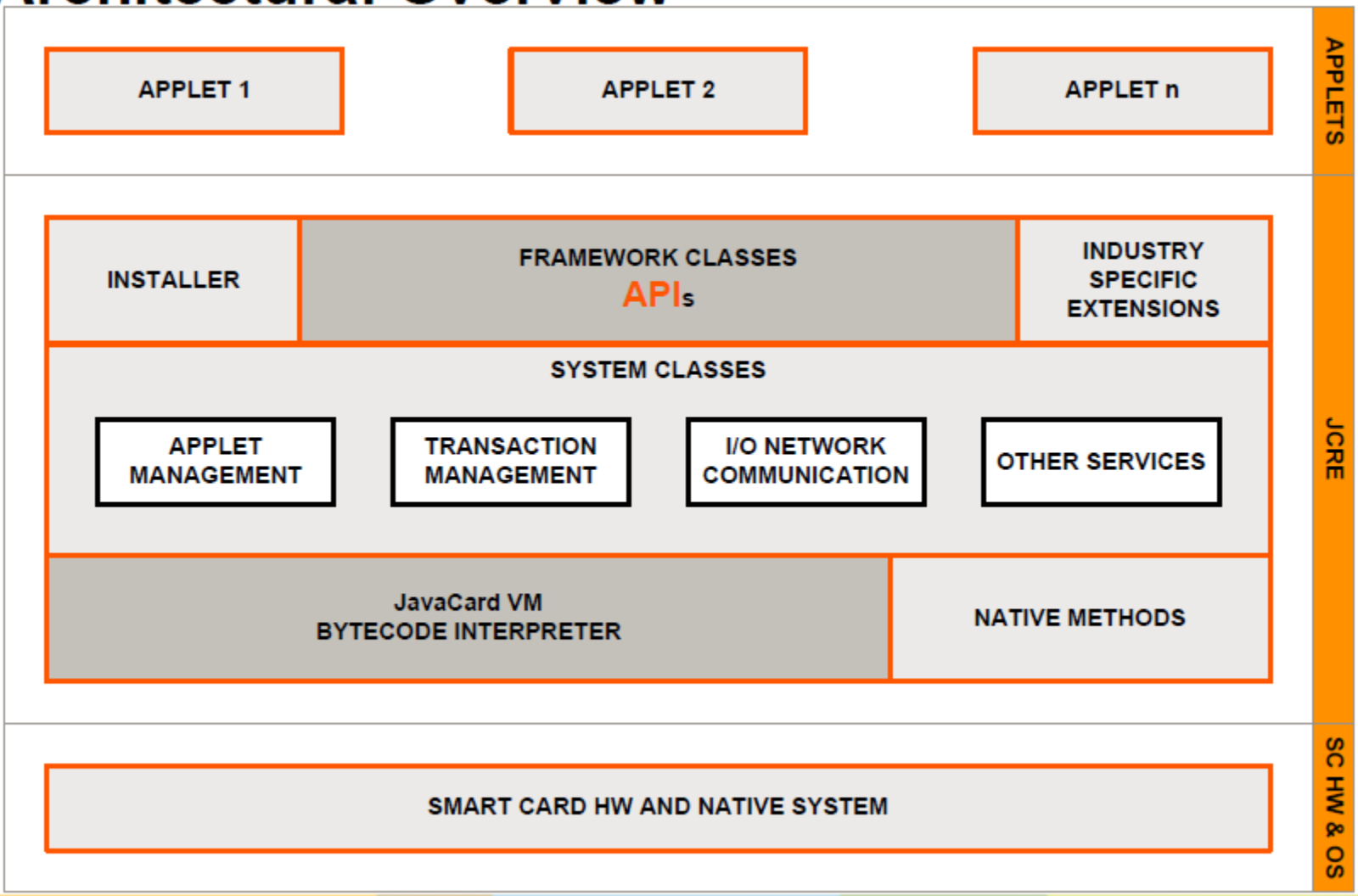




GEMPLUS

code camp

Architectural Overview

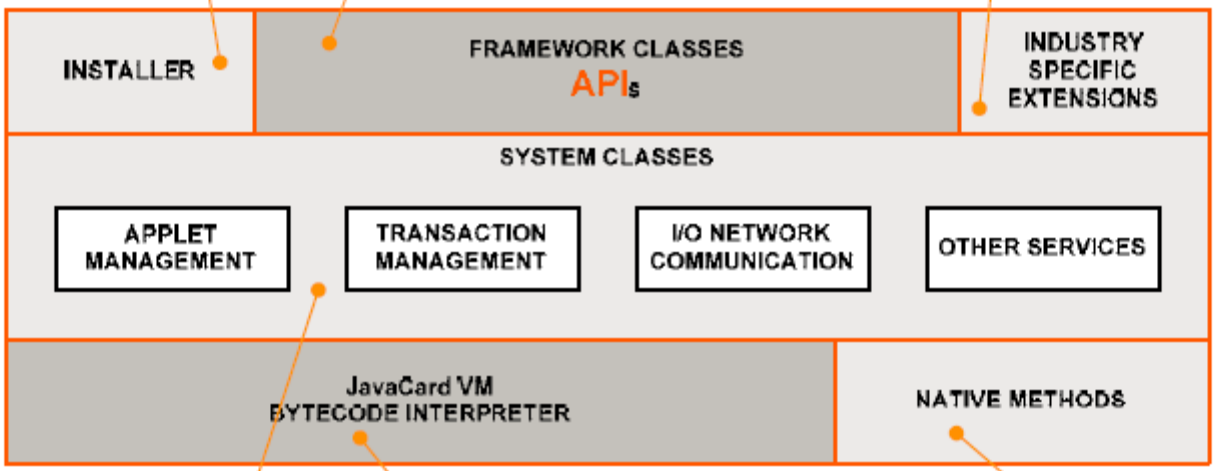


JCRE Functionality

Installer; secure post-issuance applet downloading

APIs; language, framework, security, crypto APIs for applet creation.

Additional services; i.e. Visa OpenPlatform for secure financial services

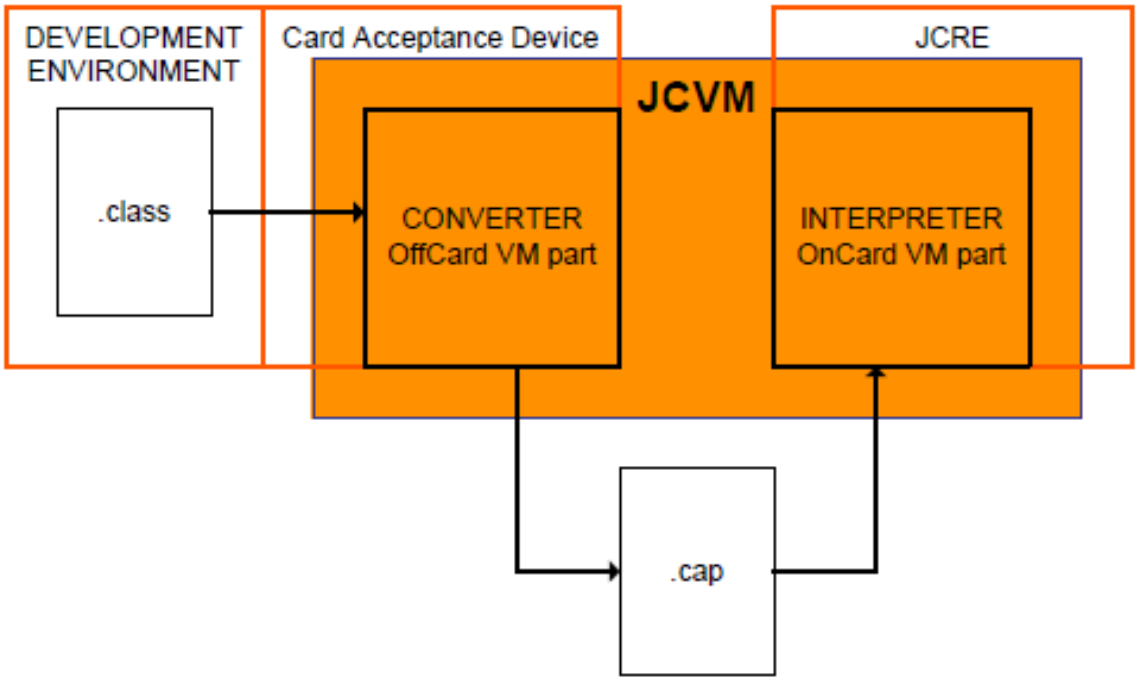


System Classes; acting as a OS core, control of transaction, communication applet creation, selection and de-selection, native method invocation.

JCVM; bytecode execution, memory allocation, object management, runtime security

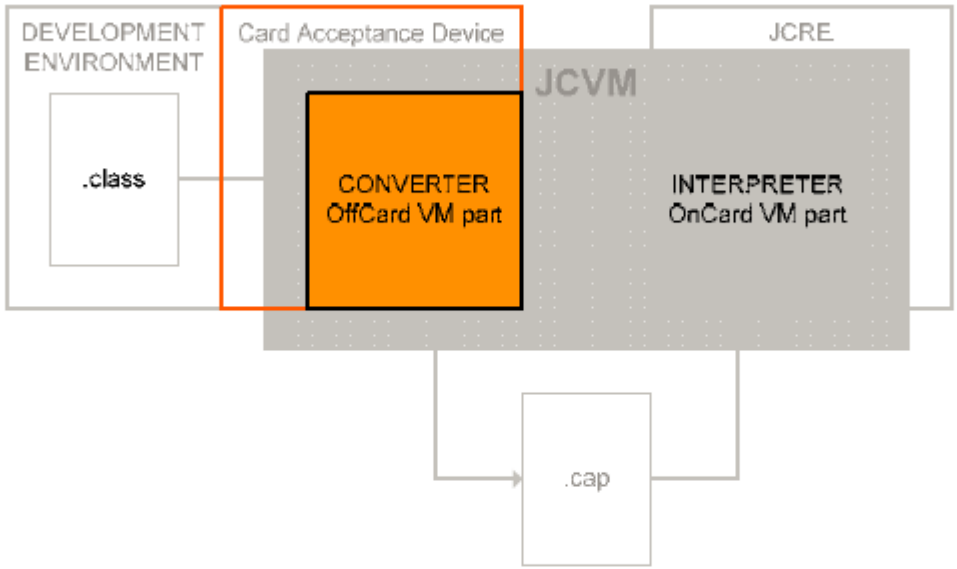
NM; JCVM support, low level comm. protocols, memory management, crypto support etc.

JavaCard Virtual Machine (JCVM)



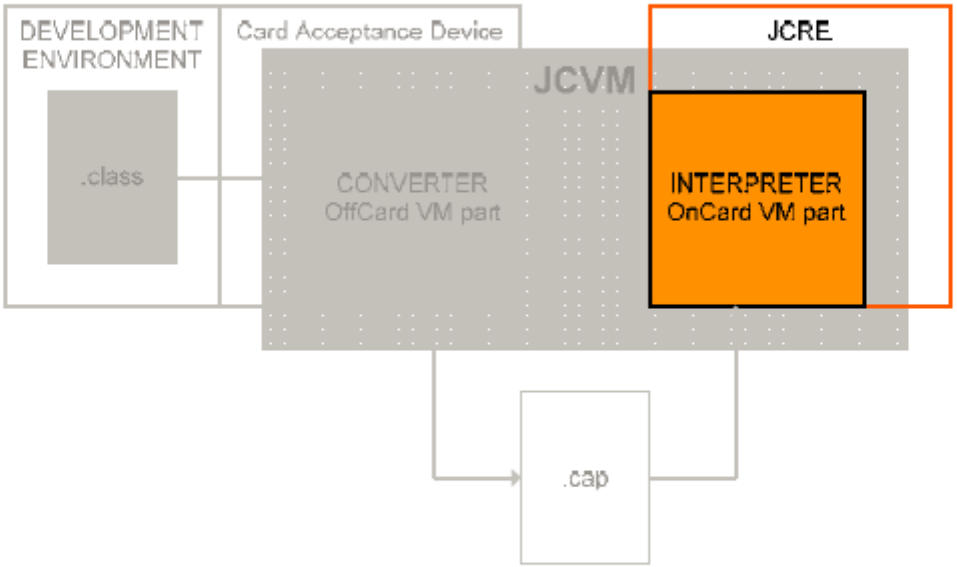
- Implemented as two separated entities

OFF-CARD JavaCard Virtual Machine (JCVM)



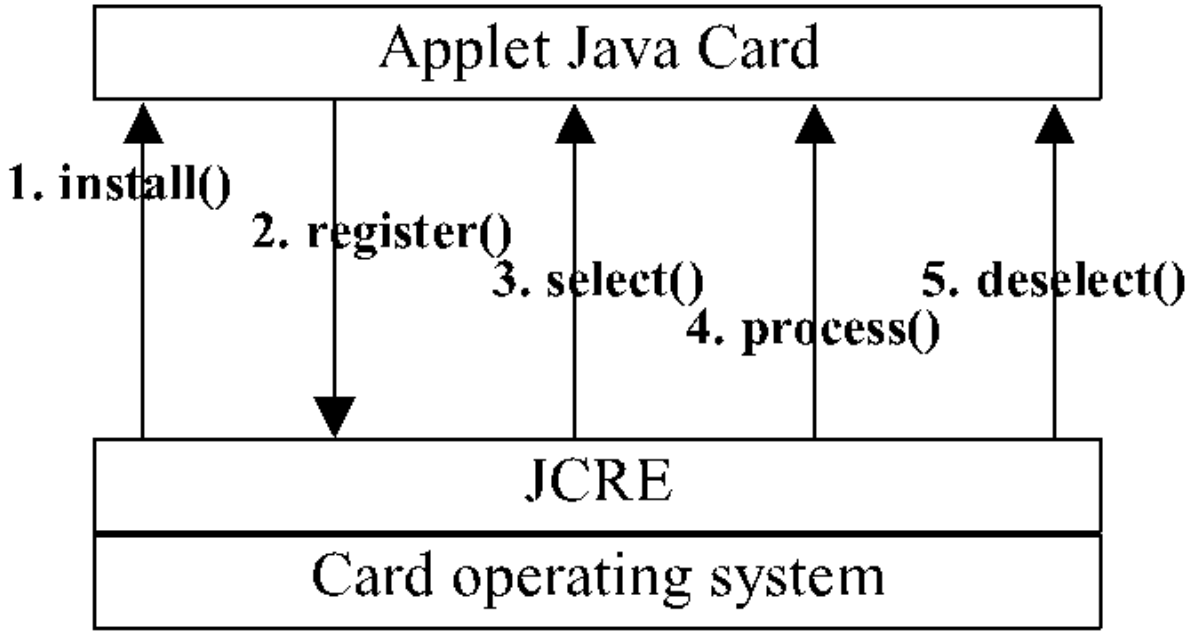
- Converts compiled java code (class files)
- Forms .cap file
- Part of development environment
- Normally related to card (Operating System dependency possible)

ON-CARD JavaCard Virtual Machine (JCVM)

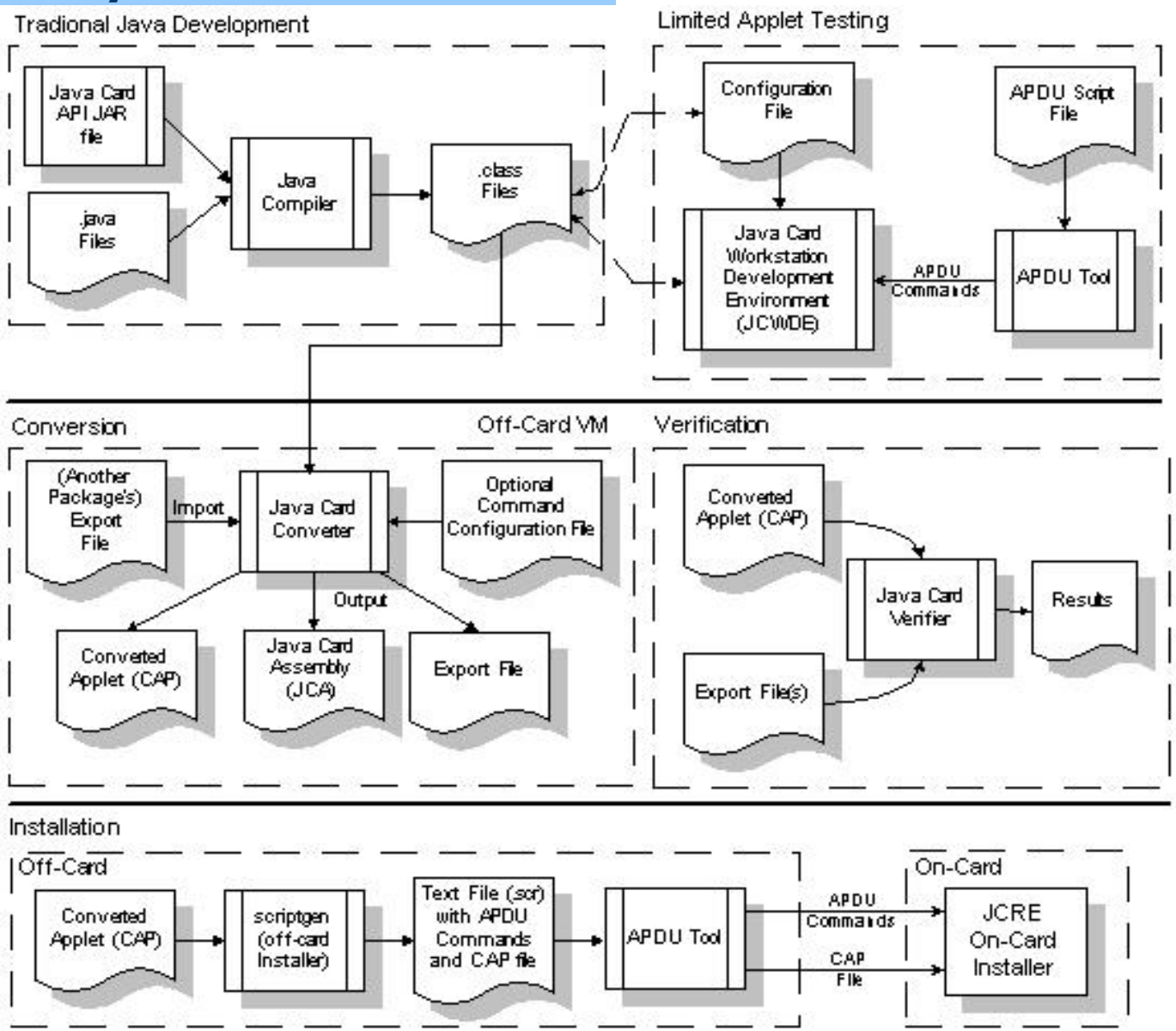


- Interpretation of the bytecode packed in .cap file
- Central entity of the JCRE
- Card performance is not only related to this part of JCVM

Java Card Life Cycle



Development Cycle



Design the APDUs

- PIN verification

Java private method	CLA 1 byte	INS 1 byte	P1 1 byte	P2 1 byte	Lc 1 byte	Data Field 5 bytes	Le
verify()	0x80	0x20	0x00	0x00	0x05	0x01 0x02 0x03 0x04 0x05	0x02

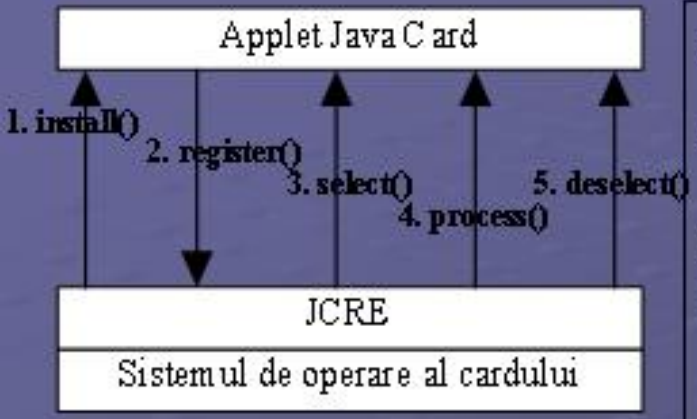
- Interrogation to find which amount of money the card contains

Java private method	CLA 1 byte	INS 1 byte	P1 1 byte	P2 1 byte	Lc 1 byte	Data Field 5 bytes	Le
getBalance()	0x80	0x50	0x00	0x00	0x00	Nothing	0x7F

- APDU command for debit – e.g. to deposit money on the card – usually executed by a bank

Java private method	CLA 1 byte	INS 1 byte	P1 1 byte	P2 1 byte	Lc 1 byte	Data Field 5 bytes	Le
debit()	0x80	0x40	0x00	0x00	0x01	0x64	0x7F

Cardlet lifecycle and source code




```
package com.sun.javacard.samples.wallet;

import javacard.framework.*;

public class Wallet extends Applet {
...
   //constructor
   private Wallet (byte[] bArray,short
bOffset,byte bLength) {...}
   //Life-cycle methods - specifice fiecarui applet
   public static void install(...) {...}
   public void select() {...}
   public void deselect() {...}
   public void process(APDU apdu) {...}
//private methods - specifice doar acestui applet
   private void credit(APDU apdu) {...}
   private void debit(APDU apdu) {...}
   private void getBalance(APDU apdu) {...}
   private void verify(APDU apdu) {...}
...
}
```

Minimal Wallet Java Card DEMO

DEMO

For Lecture – Command Line

For Labs:

Obsolete: http://sourceforge.net/projects/izynfc/

1. Oracle Java Card Classic Edition SDK & Eclipse Plug-In (https://www.oracle.com/technetwork/java/embedded/javacard/downloads/javacard-sdk-2043229.html)

2. JAVACOS (https://javacardos.com/tools)

- PAY Attention to Viruses

Build a sample application

- ***Step 1 – Compilation*** **– Positioning into the directory 'Wallet1\com\sun\javacard\samples\wallet' and subsequently introducing the commands:**
  - SET _CLASSES=.;%JC_HOME%\lib\apduio.jar;%JC_HOME%\lib\apdutool.jar;%JC_HOME%\lib\jcwde.jar;%JC_HOME%\lib\converter.jar;%JC_HOME%\lib\scriptgen.jar;%JC_HOME%\lib\offcardverifier.jar;%JC_HOME%\lib\api.jar;%JC_HOME%\lib\installer.jar;%JC_HOME%\lib\capdump.jar;%JC_HOME%\lib\javacardframework.jar;%JC_HOME%\samples\classes;%CLASSPATH%;
  - %JAVA_HOME%\bin\javac.exe -g -classpath %_CLASSES% com\sun\javacard\samples\wallet\Wallet.java
- ***Step 2 – Editing the configuration file for card uploading*** **– in the directory Wallet1 in the directory structure mentioned for step 1 it is created a text file for configuration named 'wallet.app' that includes the following:**
  - // applet AID
  - com.sun.javacard.installer.InstallerApplet 0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0x8:0x1
  - com.sun.javacard.samples.wallet.Wallet 0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1

Build application

- ***Step 3 – Conversion of the java byte code class into a binary file that can be interpreted by the JCVM on the card*** – with the configuration file from step 2, it is called in the command prompt, also from the directory Wallet1, the following instruction:
  - %JC_HOME%\bin\converter.bat -config com\sun\javacard\samples\wallet\ Wallet.opt
- ***Step 4 – uploading the binary execution file in the volatile memory of the card*** – it is run the so-called off-card installer. The first command is:
  - %JC_HOME%\bin\scriptgen.bat -o Wallet.scr com\sun\javacard\samples\wallet\javacard\wallet.cap

Build application

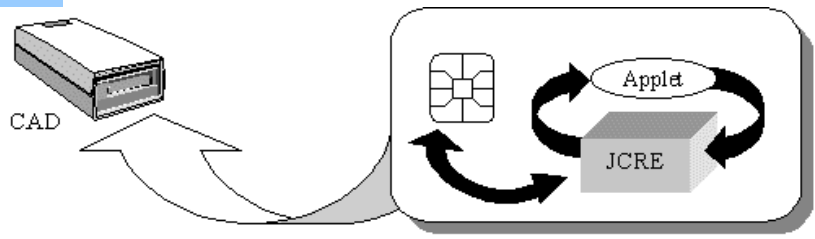
- Step 5 - So, in the new command prompt window it is wrote the command:
  - cref.exe –o eeprom1
  - The command is meant to save the EEPROM memory of the card, after it is modified it by APDU commands. The command launches the JCRE emulation, and JCRE listens to TCP/IP on the pre-defined port 9025, in order to receive APDU commands. The emulation of JCRE stops when it receives a powerdown; by the APDU command script file (extension scr).
  - Now in the old command prompt window it is run the command:
    - %JC_HOME%\bin\apdutool.bat Wallet.scr > Wallet.scr.out

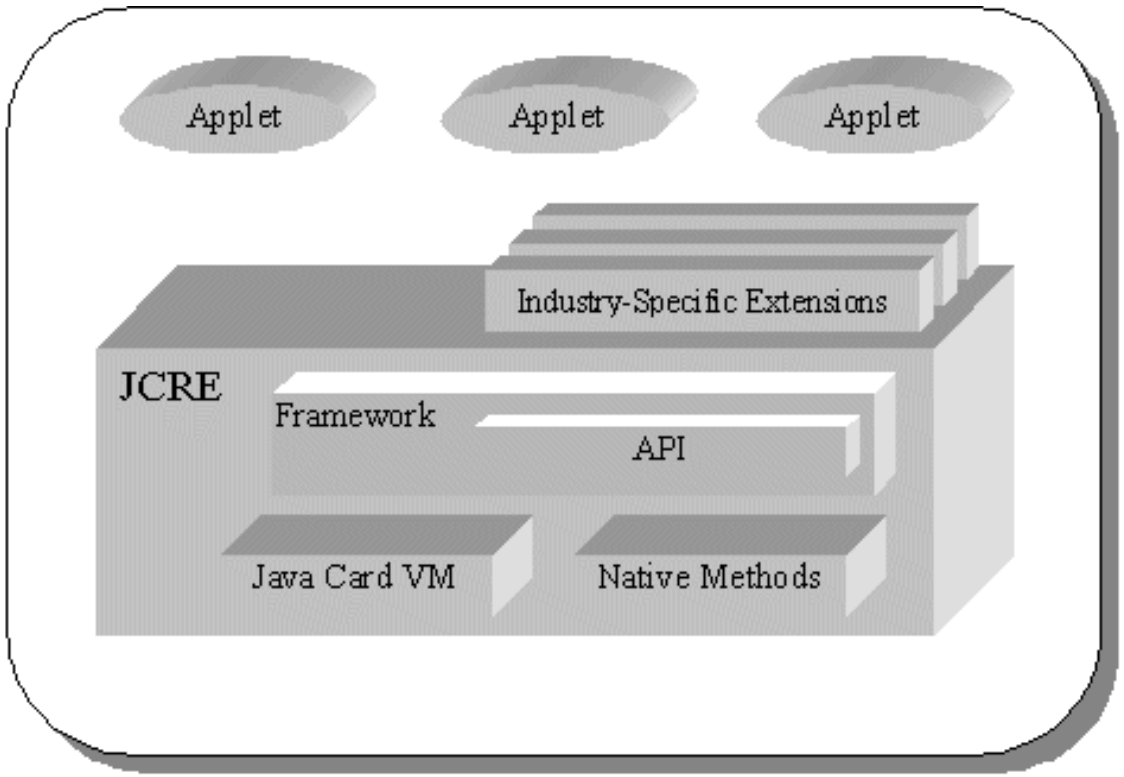
Build Application

- ***Step 6 – simulating the action between the host application and the card applet by CAD*** – it is executed after step 5. At step 5, the JCRE simulation has ended. Now it must be relaunched by the command:
  - cref.exe –i eeprom1 –o eeprom1
  - The command takes on the memory image from the eeprom1 file, modifies it according to the received APDU commands and saves it again with the modifications in the same eeprom1 file. On the same JCRE port it simulates receiving APDU commands and by the command
  - %JC_HOME%\bin\apdutool.bat demoWallet.scr > demoWallet.scr.cjcre.out
  - it is sent the APDU Commands for test and simulation of the applet, and the APDU Answers are in the file demoWallet.scr.cjcre.out.

RESULTS INTERPRETATION

JCRE & Card Session





Java Card Runtime Features

- **Persistent and Transient Objects**
  - Java objects are usually persistent and stored within EEPROM. For security reasons the TRANSIENT objects (has nothing to do with transient from Java SE) are stored in RAM
- **Atomic operations and transactions**
  - The JCVM ensures that each write operation to a single field in an object or in a class is ATOMIC. In addition, the JCRE provides transaction APIs.
- **Applet Firewall & Sharing Mechanisms**
  - The applet firewall isolates applets. Each applet runs into a designated space. In situations where applets need to share data or access JCRE services, JCVM implements sharing functions through **secure sharing mechanisms**.
- **JCRMI – Java Card Remote Method Invocation**
  - Calls methods implemented within the smart card, by sending APDUs having parameters that should be passed to the remote method.

JC VM Limitations

MAX 255 public classes and interfaces are allowed in one package

One class can implement 15 interfaces

MAX 256 public or protected static fields for one class

MAX 256 public or protected static methods for one class

ARRAY can hold MAX 32767 fields

MAX 255 local variables per method are allowed

MAX 32767 byte-codes per one method

Java Card Language Limitations & JCVM Constraints

Language Features	Dynamic class loading, security manager (`java.lang.SecurityManager`), threads, object cloning, and certain aspects of package access control are not supported.
Keywords	`native`, `synchronized`, `transient`, `volatile`, `strictfp` are not supported.
Types	There is no support for `char`, `double`, `float`, and `long`, or for multidimensional arrays. Support for `int` is optional.
Classes and Interfaces	The Java core API classes and interfaces (`java.io`, `java.lang`, `java.util`) are unsupported except for `Object` and `Throwable`, and most methods of `Object` and `Throwable` are not available.
Exceptions	Some `Exception` and `Error` subclasses are omitted because the exceptions and errors they encapsulate cannot arise in the Java Card platform.

There are also programming-model limitations. For instance a loaded library class can no longer be extended in the card; it is implicitly made `final`.

Packages	A package can refer to up to 128 other packages
	A fully qualified package name is limited to 255 bytes. Note that the character size depends on the character encoding.
	A package can have up to 255 classes.
Classes	A class can directly or indirectly implement up to 15 interfaces.
	An interface can inherit from up to 14 interfaces.
	A package can have up to 256 static methods if it contains applets (an *applet package*), or 255 if it doesn't (a *library package*).
	A class can implement up to 128 public or protected instance methods, and up to 128 with package visibility.

In the Java Card VM, as in the J2SE VM, .class files are central, but the JCVM specification defines two other file formats to further optimize

Section II – Java Card Technology

Java Card API

The Java Card API specification defines a small subset of the traditional Java programming language API - even smaller than that of J2ME's CLDC. There is no support for `Strings`, or for multiple threads. There are no wrapper classes like `Boolean` and `Integer`, and no `Class` or `System` classes.

In addition to its small subset of the familiar Java core classes the Java Card Framework defines its own set of core classes specifically to support Java Card applications. These are contained in the following packages:

- `java.io` defines one exception class, the base `IOException` class, to complete the RMI exception hierarchy. None of the other traditional java.io classes are included.
- `java.lang` defines `Object` and `Throwable` classes that lack many of the methods of their J2SE counterparts. It also defines a number of exception classes: the `Exception` base class, various runtime exceptions, and `CardException`. None of the other traditional `java.lang` classes are included.
- `java.rmi` defines the `Remote` interface and the `RemoteException` class. None of the traditional `java.rmi` classes are included. Support for Remote Method Invocation (RMI) is included to simplify migration to, and integration with, devices that use Java Card technology.
- `javacard.framework` defines the interfaces, classes, and exceptions that compose the core Java Card Framework. It defines important concepts such as the Personal Identification Number (`PIN`), the Application Protocol Data Unit (`APDU`), the Java Card applet (`Applet`), the Java Card System (`JCSystem`), and a utility class. It also defines various `ISO7816` constants and various Java Card-specific exceptions. Table 5 summarizes this package's contents:

Java Card API – javacard.framework.*

Interfaces	`ISO7816` defines constants related to ISO 7816-3 and ISO 7816-4.
	`MultiSelectable` identifies applets that can support concurrent selections.
	`PIN` represents a personal identification number used for security (authentication) purposes.
	`Shareable` identifies a shared object. Objects that must be available through the applet firewall must implement this interface.
Classes	`AID` defines an ISO7816-5-conforming Application Identifier associated with an application provider; a mandatory attribute of an applet.
	`APDU` defines an ISO7816-4-conforming Application Protocol Data Unit, which is the communication format used between the applet (on-card) and the host application (off-card).
	`Applet` defines a Java Card application. All applets must extend this abstract class.
	`JCSystem` provides methods to control the applet life-cycle, resource and transaction management, and inter-applet object sharing and object deletion.
	`OwnerPIN` is an implementation of the `PIN` interface.
	`Util` provides utility methods for manipulation of arrays and shorts, including `arrayCompare()`, `arrayCopy()`, `arrayCopyNonAtomic()`, `arrayFillNonAtomic()`, `getShort()`, `makeShort()`, `setShort()`.
Exceptions	Various Java Card VM exception classes are defined: `APDUException`, `CardException`, `CardRuntimeException`, `ISOException`, `PINException`, `SystemException`, `TransactionException`, `UserException`.

Java Card API – javacard.framework.service.*

Interfaces	`Service`, the base service interface, defines the methods `processCommand()`, `processDataIn()`, and `processDataOut()`.
	`RemoteService` is a generic `Service` that gives remote processes access to services on the card.
	`SecurityService` extends the `Service` base interface, and provides methods to query the current security status, including `isAuthenticated()`, `isChannelSecure()`, and `isCommandSecure()`.
Classes	`BasicService` is a default implementation of a `Service`; it provides helper methods to handle APDUs and service collaboration.
	`Dispatcher` maintains a registry of services. Use a dispatcher if you want to delegate the processing of an APDU to several services. A dispatcher can process an APDU completely with the `process()` method, or dispatch it for processing by several services with the `dispatch()` method.
Exceptions	`ServiceException` a service-related exception.

Java Card API – javacard.security.*

Interfaces	Generic base interfaces `Key`, `PrivateKey`, `PublicKey`, and `SecretKey`, and subinterfaces that represent various types of security keys and algorithms: `AESKey`, `DESKey`, `DSAKey`, `DSAPrivateKey`, `DSAPublicKey`, `ECKey`, `ECPrivateKey`, `ECPublicKey`, `RSAPrivateCrtKey`, `RSAPrivateKey`, `RSAPublicKey`
Classes	`Checksum`: abstract base class for CRC algorithms
	`KeyAgreement`: base class for key-agreement algorithms
	`KeyBuilder`: key-object factory
	`KeyPair`: a container to hold a pair of keys, one private, one public
	`MessageDigest`: base class for hashing algorithms
	`RandomData`: base class for random-number generatorss
	`Signature`: base abstract class for signature algorithms
Exceptions	`CryptoException`: encryption-related exceptions such as unsupported algorithm or uninitialized key.

- `javacardx.crypto` is an extension package that defines the interface `KeyEncryption` and the class `Cypher`, each in its own package for easier export control. Use `KeyEncryption` to decrypt an input key used by encryption algorithms. `Cypher` is the base abstract class that all ciphers must implement.

- `javacardx.rmi` is an extension package that defines the Java Card RMI classes. It defines two classes, `CardRemoteObject` and `RMIService`. `CardRemoteObject` defines two methods, `export()` and `unexport()`, to enable and disable remote access to an object from outside the card. `RMIService` extends `BasicService` and implements `RemoteService` to process RMI requests.

Java Card Technology
Applet Firewall and Object Sharing

Partial Copyright – only for Object Sharing

Instructors:

Fu-Chiung Cheng

(鄭福炯)

Associate Professor

Computer Science & Engineering

Tatung University

Applet Firewall and Object Sharing

This section explains the behavior of objects, exceptions, and Applets in the presence of their firewall and discusses how applets can safely share data by using the Java Card APIs.

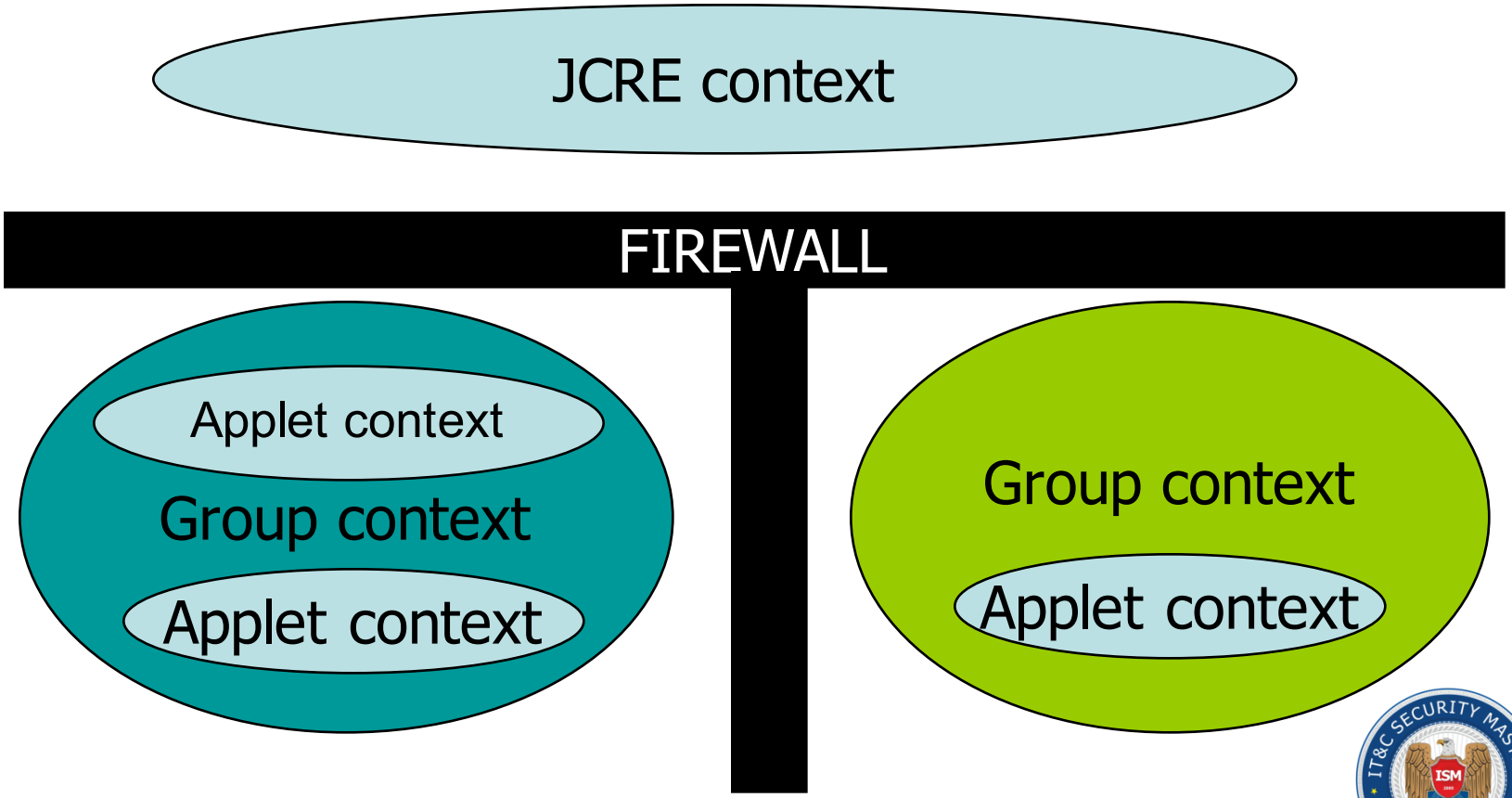
Applet Firewall and Object Sharing

What is the Context?

- The applet firewall partitions the Java Card object system into separate protected object spaces called context.
- When an applet instance is created, the JCRE assigns it a context which is essentially a group context.
- All applet instances of a single Java package share the same group context.
- There is no firewall between two applet instances in a group context.
- The JCRE maintain its own JCRE context
- JCRE context has special privileges:
  - Access from the JCRE context to any applet's context

Applet Firewall and Object Sharing

What is the Context?



Applet Firewall and Object Sharing

Object ownership

- At any time, there is only one active context within the virtual machine: either the JCRE context or an applet's group context.
- When a new object is created, it is assigned an owning context-----the currently active context.

Object Access Exception

- If the contexts do not match, the access is denied, and the comparison results in a SecurityException.

Applet Firewall and Object Sharing

Static Fields and Methods

- Only instances of classes---objects---are owned by context; classes themselves are not.
- Static fields and methods are accessible from any applet context in the defining package (e.g. group context).

Object Access across Context

- Sharing mechanisms are accomplished by the following means:
  - JCRE privileges
  - JCRE entry point objects
  - Global arrays
  - Shareable interfaces

Applet Firewall and Object Sharing

Context Switch

- When a sharing mechanism is applied, the Java Card virtual machine enables access by performing a context switch.
- Context switches occur:
  - only during invocation of and return from instance methods of an object owned by a different context,
  - during exception exits form those methods.
- During a context-switching method invocation, the current context is saved, and the new context become the currently active context.
- When the virtual machine begins running after card reset, the JCRE context is always the currently active context.

Applet Firewall and Object Sharing

JCRE Privileges

- JCRE Privileges permits to the JCRE from the card to:
  - invoke a method on any object or
  - access an instance field of any on the card.
- Such system privileges enable the JCRE to control system resources and manage applets
  - For example, when the JCRE receives an APDU command, it invokes the currently selected applet's select, deselect or process method
- When JCRE invokes an applet's method, the JCRE context is switched to the applet's context.
- The applet now takes control and loses the JCRE privileges.
- Any objects created after the context switch are owned by the applet.

Applet Firewall and Object Sharing

JCRE entry point objects

- By using JCRE entry point object, non-privileged users can request system services that are performed by privileged system routines.
- JCRE entry point objects are normal objects owned by the JCRE context, *but they have been flagged as containing entry point methods*.
- The entry point designation allows the public methods of such objects to be invoked from any context.
- *When that occurs, a context switch to the JCRE context is performed.*
- Notice that only the public methods of JCRE entry point objects are accessible through the firewall.
- The fields of these objects are still protected by the firewall.

JCRE entry point objects

Two categories of JCRE EPOs:

- Temporary JCRE entry point objects:
  - Sample: The APDU object and all JCRE-owned exception objects.
  - Reference to these objects can't be stored in class variables.
- Permanent JCRE entry point objects:
  - Examples :The JCRE-owned AID instances.
  - Reference to these objects can be stored and freely used.

Global Arrays

- Global arrays essentially provide a shared memory buffer whose data can be accessed by any applets and by the JCRE.
- Global arrays are a special type of JCRE entry point object.
- The applet firewall enables public fields of such arrays to be accessed from any context.
- Only primitive arrays can be designated as global and *Only JCRE can designate global arrays*.
- The only global arrays required in the Java Card APIs are the *APDU buffer* and the *byte array parameter in an applet's install method*.
- Whenever an applet is selected or before JCRE accepts a new APDU command, JCRE clears the APDU buffer.
  - No leaked message

Applet Firewall and Object Sharing

Sharing between JCRE and applets

- JCRE can access any object due to its privileged nature.
- Applet gains access to system service via JCRE entry point objects.
- JCRE and applets share primitive data by using designated global arrays.

Shareable Interface

- Shareable interface enables object sharing between applets.
- Simply an interface that extends, either directly or indirectly, the tagging interface *javacard.framework.Shareable*

public interface Shareable{}

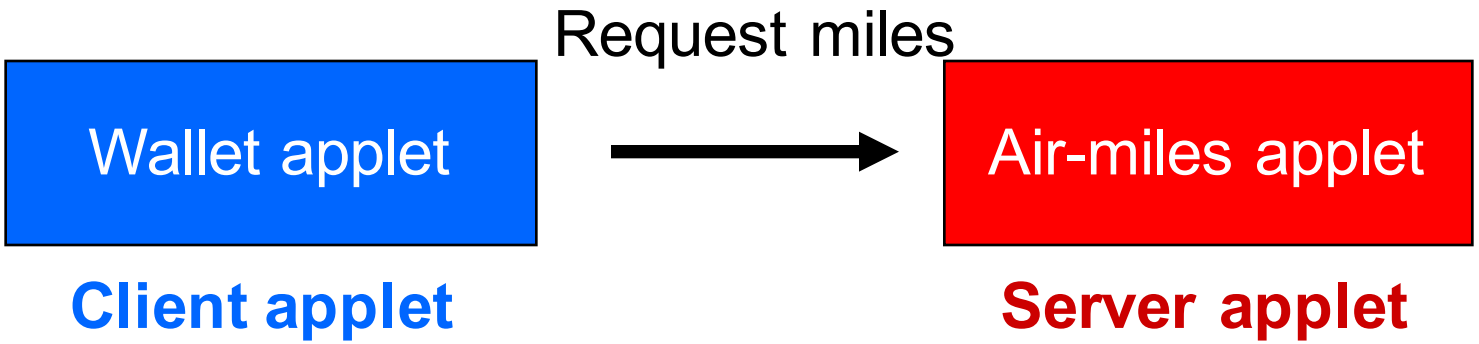
Applet Firewall and Object Sharing

SIO – Shareable Interface Object

- An object of a class that implements a shareable interface is called a SIO.
- To the owning context, an SIO is a normal object whose fields and methods can be accessed.
- To any other context, the SIO is an instance of the shareable interface type, and only the methods defined in the shareable interface are accessible.

Applet Firewall and Object Sharing

Shareable Interface Sample



Applet Firewall and Object Sharing

Shareable Interface Sample

```
package com.fasttravel.airmiles;
import javacard.framework.Shareable;
public interface AirMilesInterface extends Shareable {
    public void grantMiles(short amout);
}
```

```
package com.fasttravel.airmiles;

import javacard.framework.Shareable;

public class AirMilesApp extends Applet implements AirMilesInterface {
        private short miles;

        public void grantMiles(short amout) {
            miles = (short)( miles +  amout );
        }
    }
```

Applet Firewall and Object Sharing

Review about AID and Register

protected final void register();

protected final void register(byte[] Array, short bOffset, byte bLength)

The JCRE encapsulates the AID bytes in an AID object (owned by the JCRE) and associates this AID object with the applet. During the object sharing, this AID object is used by a client applet to specify the server.

Applet Firewall and Object Sharing

Request a SIO – Shareable Interface Object

- Client applet lookups the server AID by calling *JCSystem.lookupAID(…)* method:

```
public static AID lookupAID(byte[] buffer, short offset, byte length)
```

- Client applet gets the server SIO by calling *JCSystem.getAppletSharableInterface(…)* method:

```
public static Shareable getAppletShareableInterfaceObject(AID
server_aid, byte parameter)
```

- JCRE invokes the server applet's *getSharableInterfaceObject(…)* method

```
public Shareable getShareableInterfaceObject(AID client_aid, byte
parameter)
```

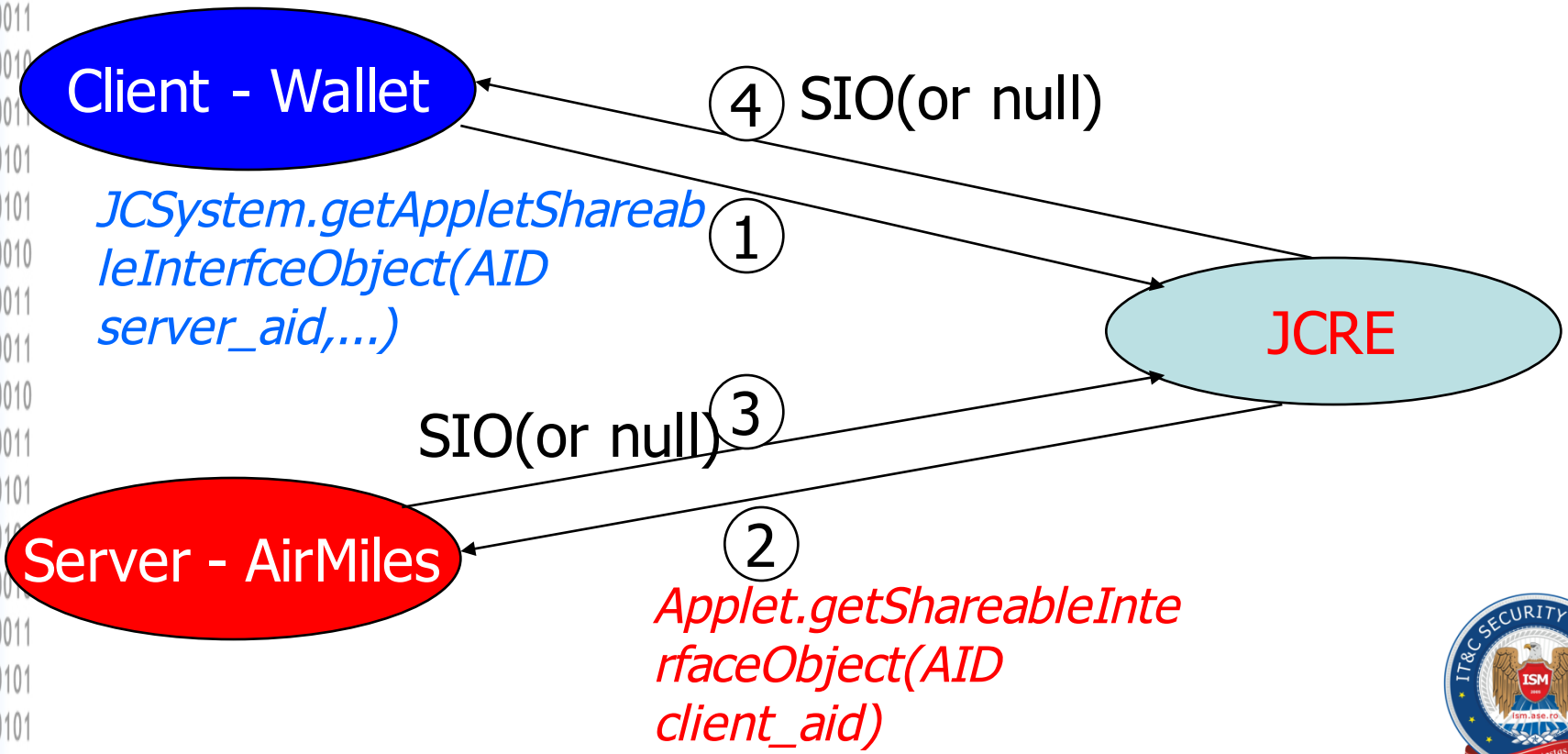
Applet Firewall and Object Sharing

Request a SIO – Shareable Interface Object

```
package com.fasttravel.airmiles;

import javacard.framework.Shareable;

public class AirMilesApp extends Applet implements AirMilesInterface {
        private short miles;

        public void grantMiles(short amout) {
            miles = (short)( miles +  amout );
        }

  public Shareable getShareableInterfaceObject (AID client_aid, byte parameter)
     {
     //authenticate the client – explained later
     return this; // return shareable interface object
     }
} //end class
```

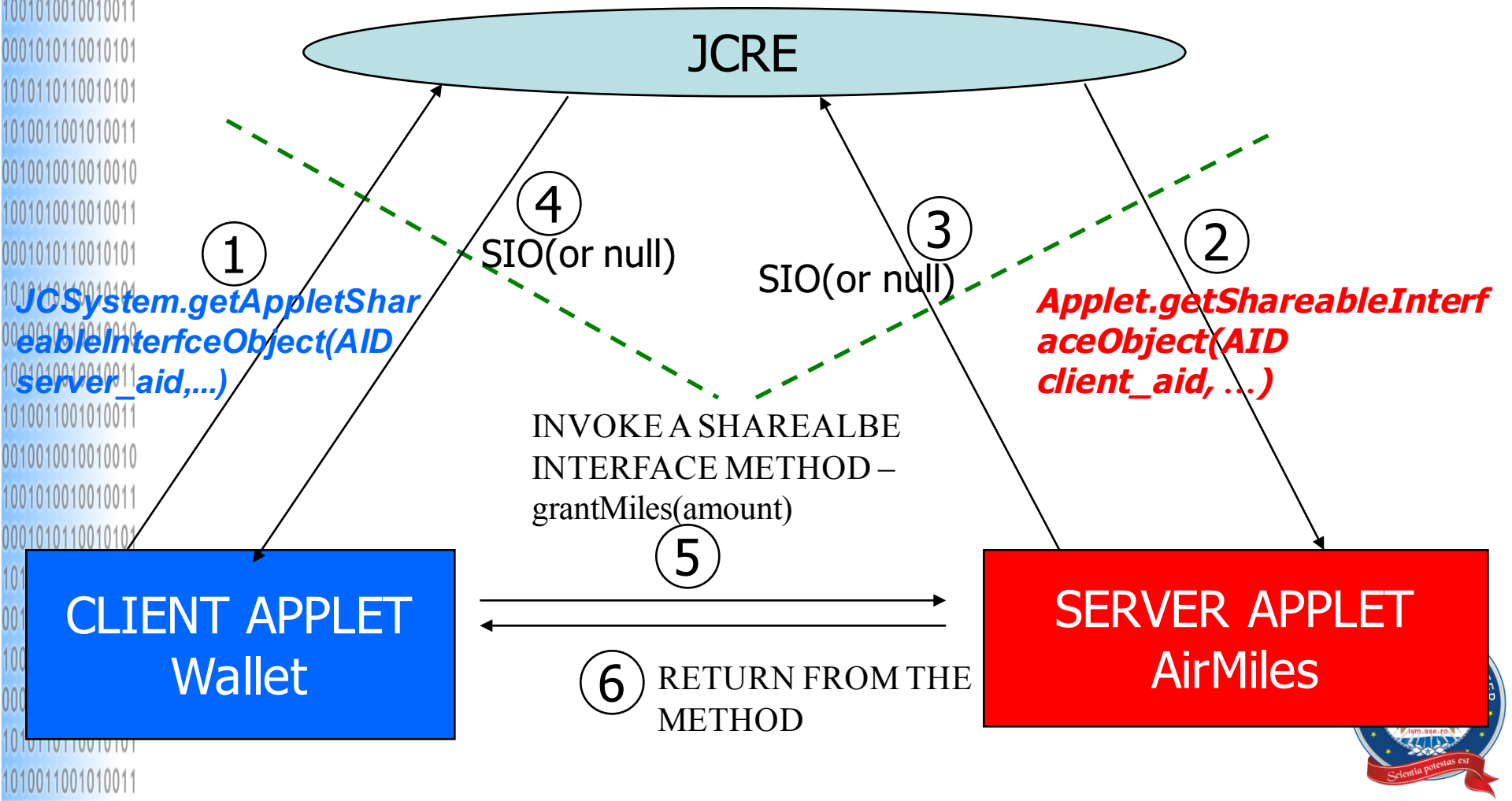
Applet Firewall and Object Sharing

Request a SIO – Shareable Interface Object



Applet Firewall and Object Sharing

Context Switches during Object Sharing



Applet Firewall and Object Sharing

Authenticate the Client Applet

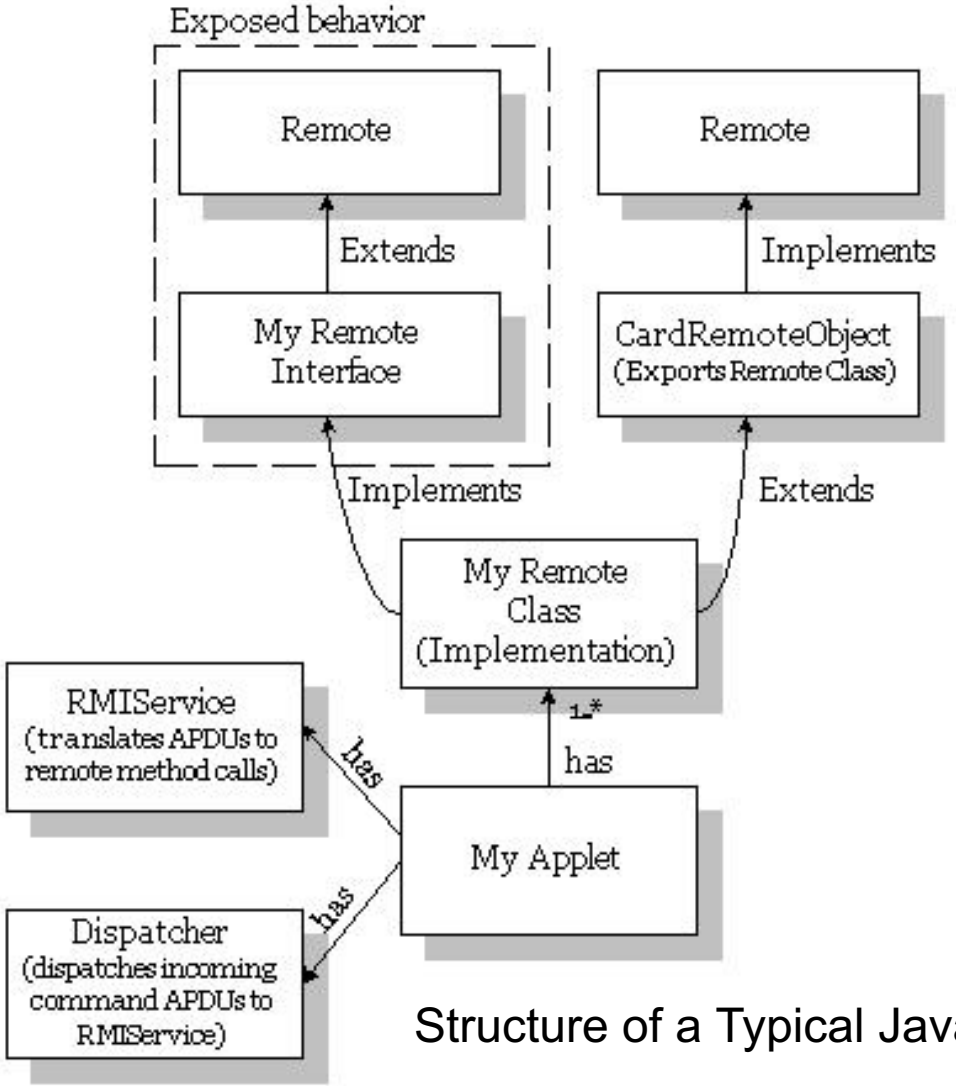
```
package com.fasttravel.airmiles;

import javacard.framework.Shareable;

public class AirMilesApp extends Applet implements AirMilesInterface {
        private short miles;

        public void grantMiles(short amout) {
            miles = (short)( miles +  amout );
        }

  public Shareable getShareableInterfaceObject (AID client_aid, byte parameter) {
     if (client_aid.equals (wallet_app_aid_bytes, (short)0, (byte)
     wallet_app_aid_bytes.length) ) == false)
                return null;
      if ( parameter != SECRET)
                return null;
       return (this); // return shareable interface object
     }
} //end class
```

Applet Firewall and Object Sharing

Verify the Client Applet again!

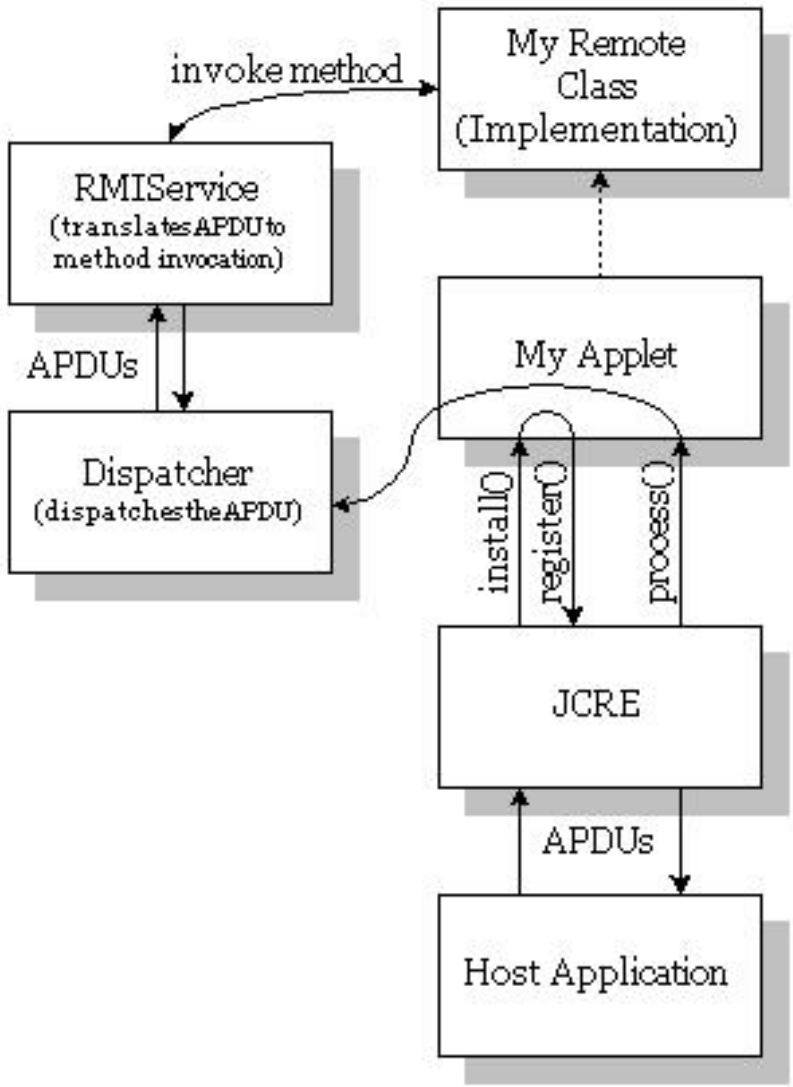
```
package com.fasttravel.airmiles;
import javacard.framework.Shareable;

public class AirMilesApp extends Applet implements AirMilesInterface {
        private short miles;

        public void grantMiles(short amout) {
            AID client_aid = JCSystem.getPreviousContextAID();
             if (client_aid.equals(wallet_app_aid_bytes, (short)0, (byte)
    wallet_app_aid_bytes.length )) == false)
                        ISOException.throwIt(SW_UNAUTHORIZED-CLIENT);

             miles = (short)( miles +  amout );
        }

 public Shareable getShareableInterfaceObject (AID client_aid, byte parameter) {
    if (client_aid.equals (wallet_app_aid_bytes, (short)0, (byte) wallet_app_aid_bytes.length) )
     == false)
              return null;
     if ( parameter != SECRET)
              return null;
      return (this); // return shareable interface object
    }
} //end class
```

Java Card Remote Method Invocation



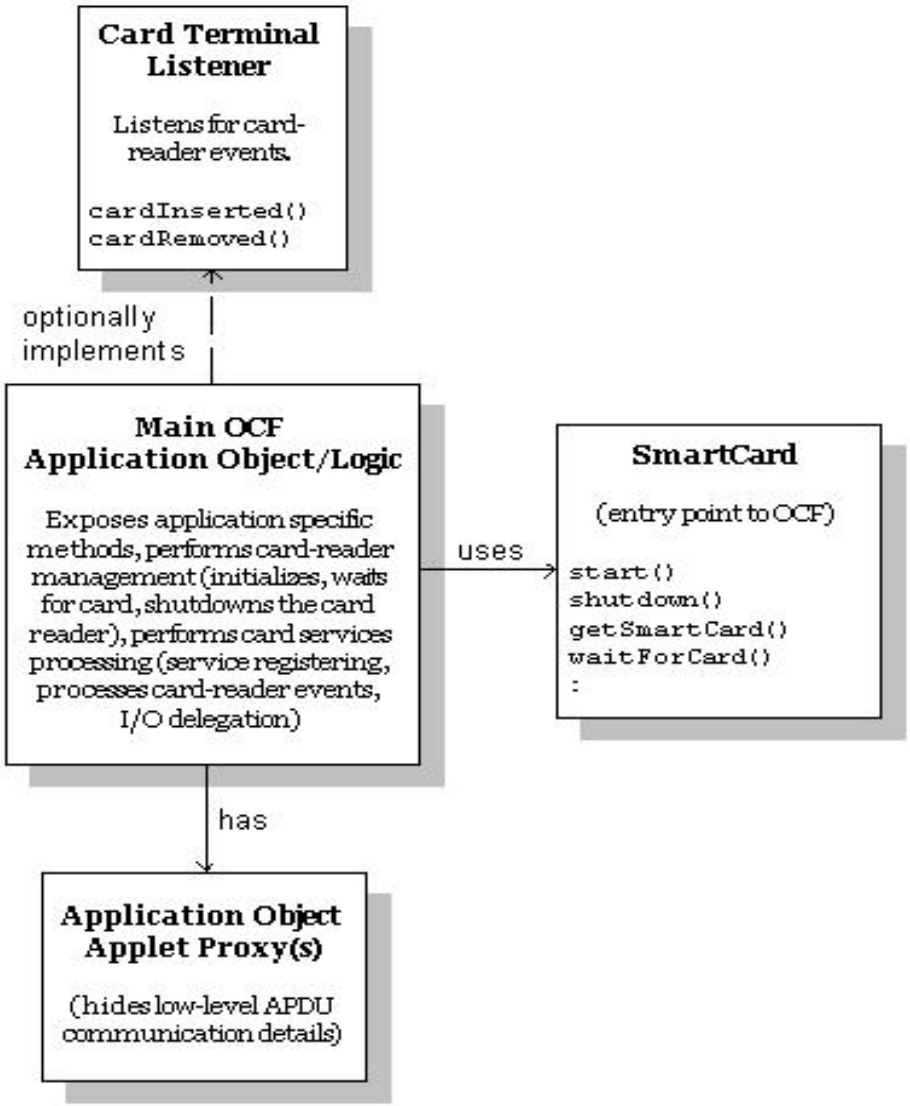
Structure of a Typical Java Card RMI Applet

Java Card Remote Method Invocation



Flow of a Java Card RMI-Based Applet

Java Card Remote Method Invocation



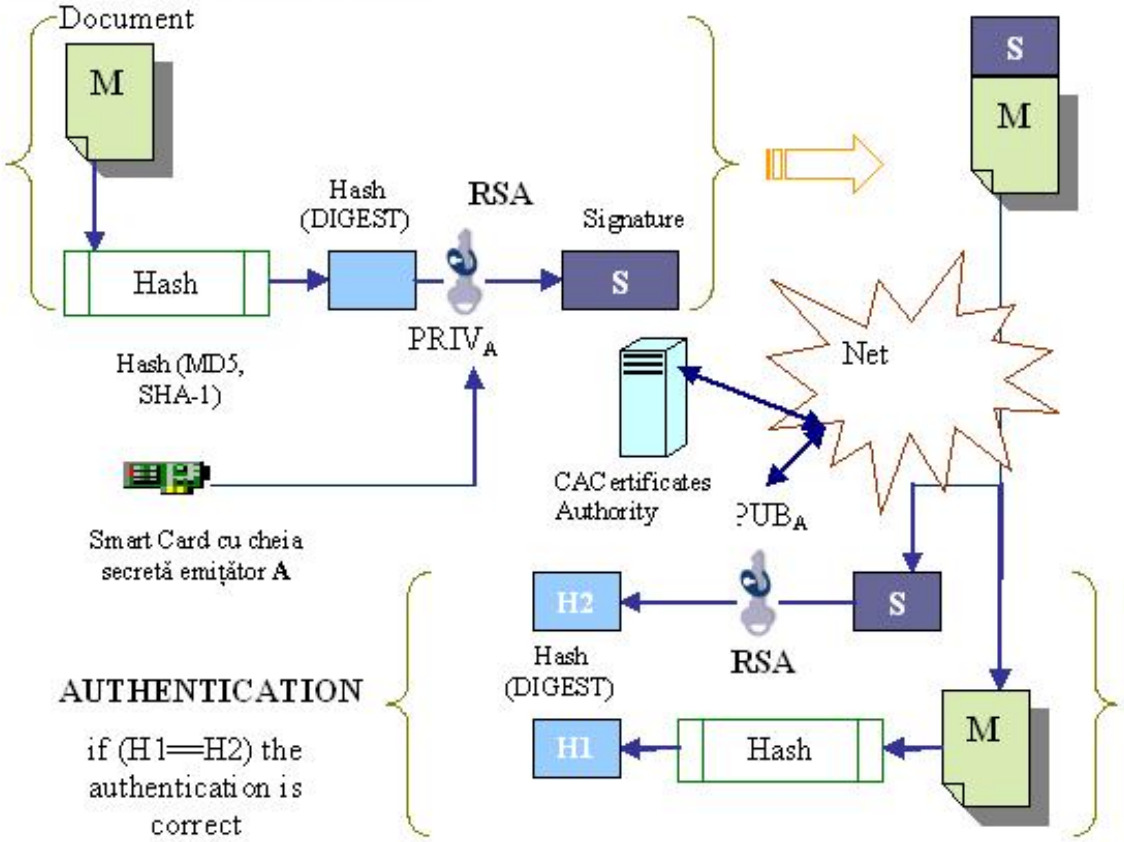
Structure of an OCF Application

Advantages of Java Smart-Card Technology

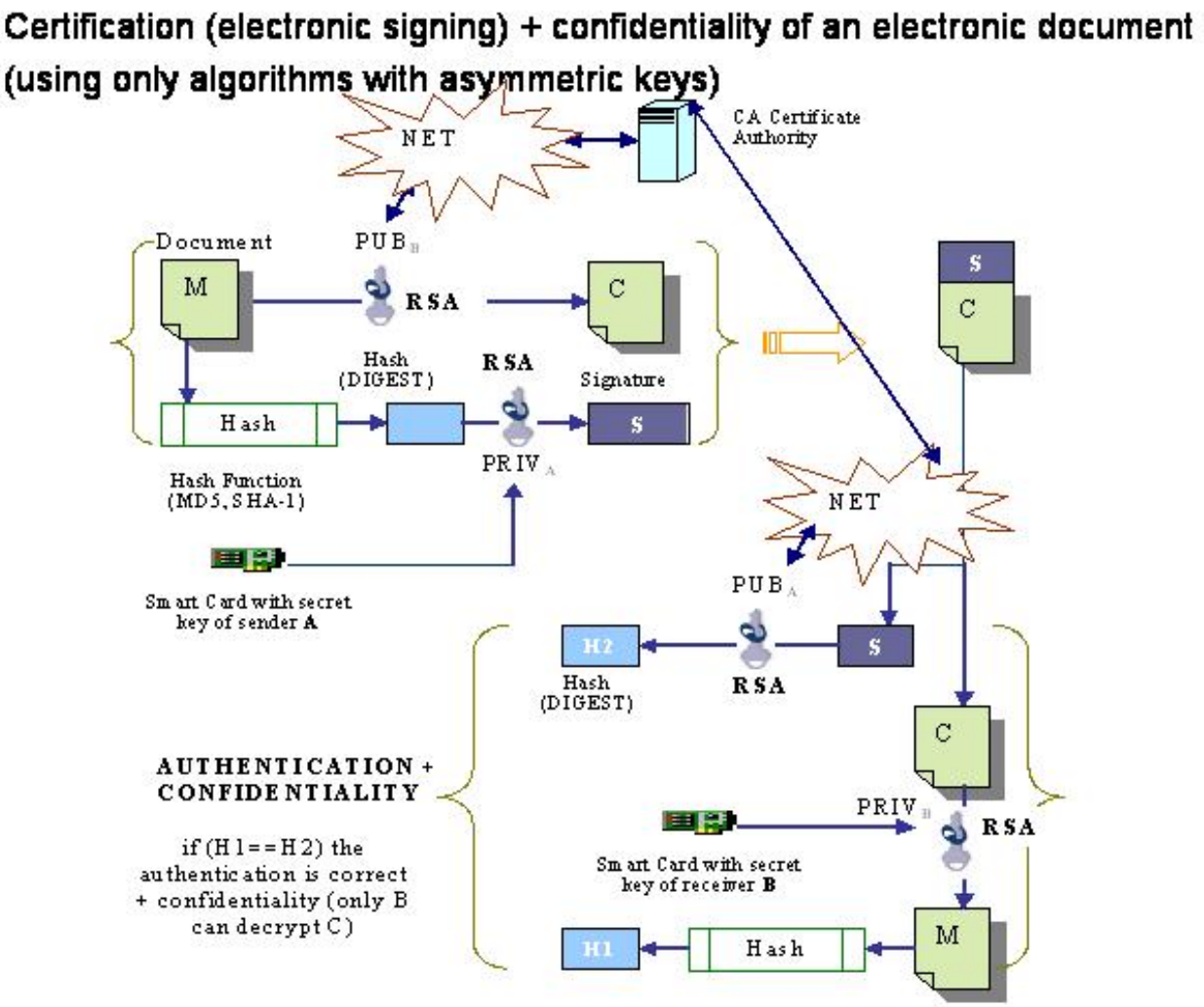
- **Interoperability** – once the applet is ready to run in a Java card platform, this applet can run on any card that have java card platform. This means if a company wants to modify the information system that they used is not necessary to buy another brand of cards;
- **Security** – is inherited from the structure of object oriented language Java. Plus, there are standard libraries for cryptography which can be used in order to offer high degree of security;
- **Scalability** – many applets can stay in the same time on the same card in secure manner. If the developed information system request a new application, simply the new applets is load on the smart card;
- **Dynamicity** – connected with the needs of end-users, can be upload on the card new applets even after this one is on the market;
- **Compatibility with existing standards** – Java platform do not eliminate any of existent standard for smart-cards, like ISO7816 or EMV – Euro-Master-Visa. The java smart-card do not depend by physical link with the card reader, if is made through ISO/IEC 14443-4:2001 or not, and do not depend of the microprocessor or chip memory from the smart card;
- **Transparency** – all specifications are made public and are for free, also the development tools from the web site.

Java Card for Signing App

Certification (electronic signing) of an electronic document.

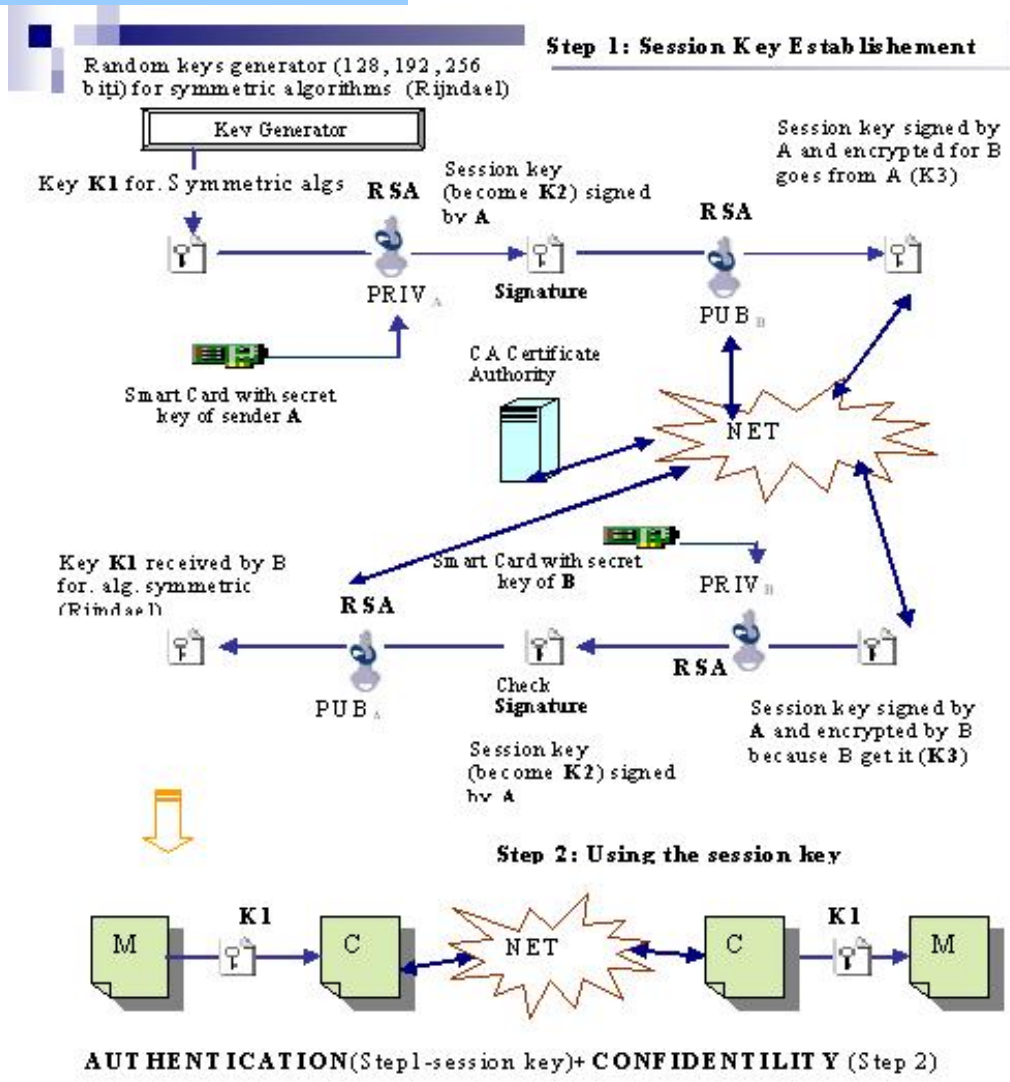


Java Card for Signing App



Section III – E-Commerce & E-Payment Smart Card Integration

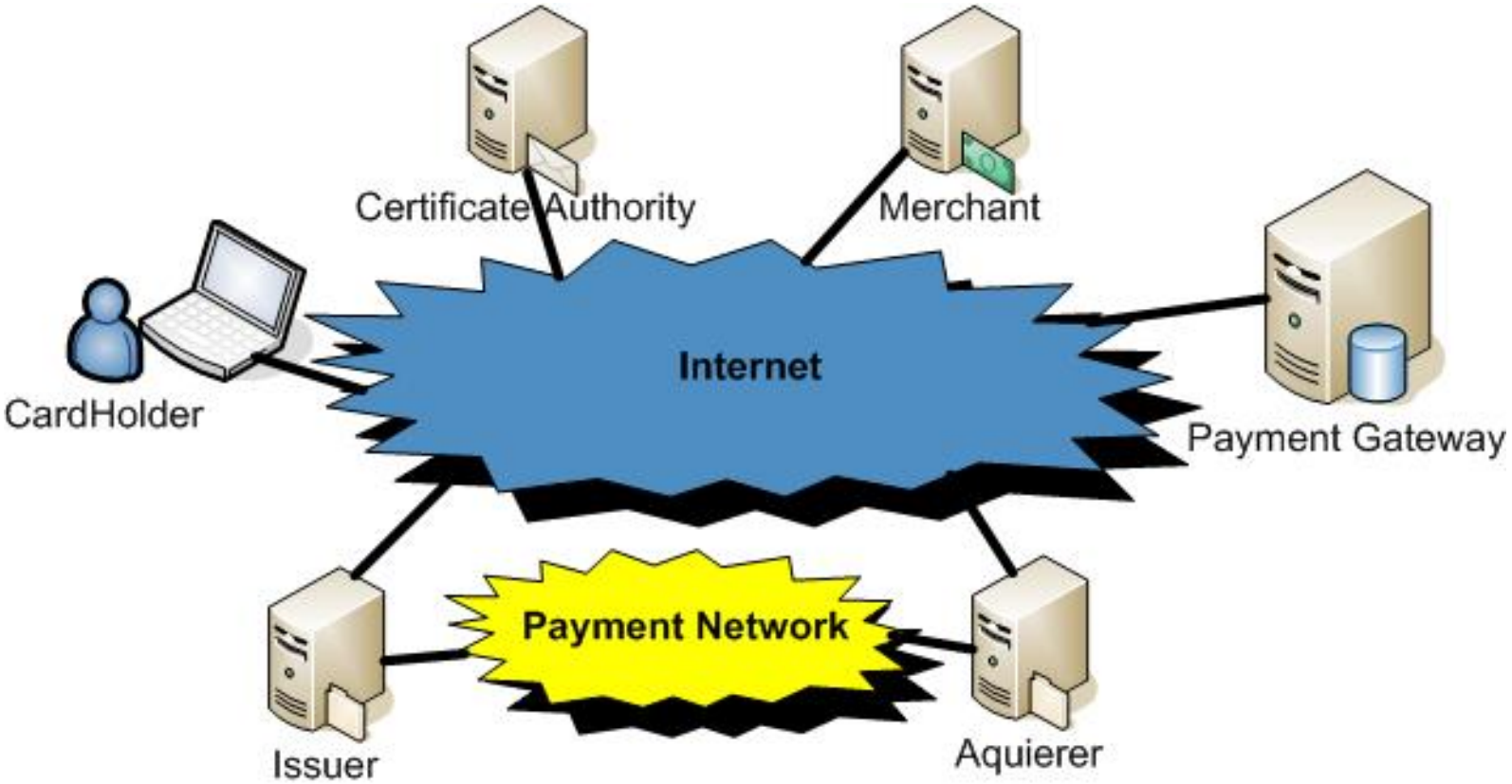
Java Card for Signing App



Certification (electronic signing) of the session key and the confidentiality of an electronic document using a hybrid cryptographic system (i.e. algorithms with symmetric key – session keys – and also asymmetric key)

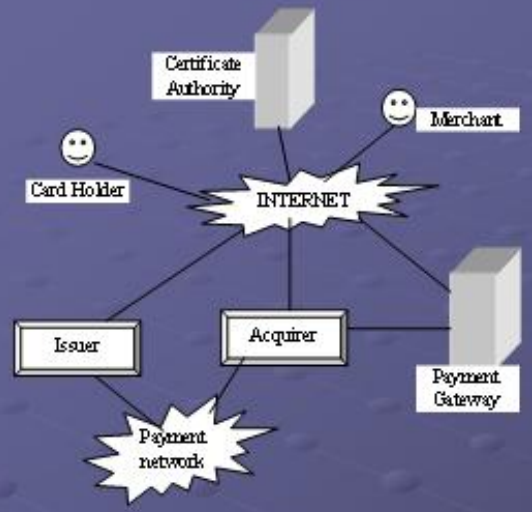
Section III – E-Commerce & E-Payment Smart Card Integration

Java Card for SET App



Java Card for SET App

SET – Secure Electronic Transaction



The SET Participants as they are described in original document specification are:

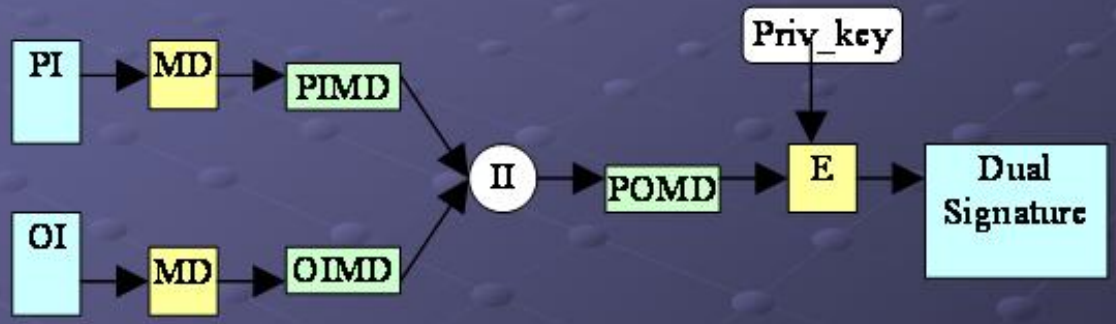
- ***Cardholder*** – an authorized holder of the credit card issued by the issuer;
- ***Merchant*** – a person who has some goods/services to sell;
- ***Issuer*** – a financial institution that issues the credit card;
- ***Acquirer*** – a financial institution that establishes an account with the merchant and process payment card authorizations and payments. It provide the interface between multiple issuers and a merchant so that the merchant does not need to deal with multiple issuers;
- ***Payment gateway*** – connected to the acquirer, the payment gateway interfaces between SET and existing payment networks for carrying out payment functions;
- ***Certification authority*** – an trusted authority which issues X.509v3 certificates.

Java Card for SET App

Dual Signature

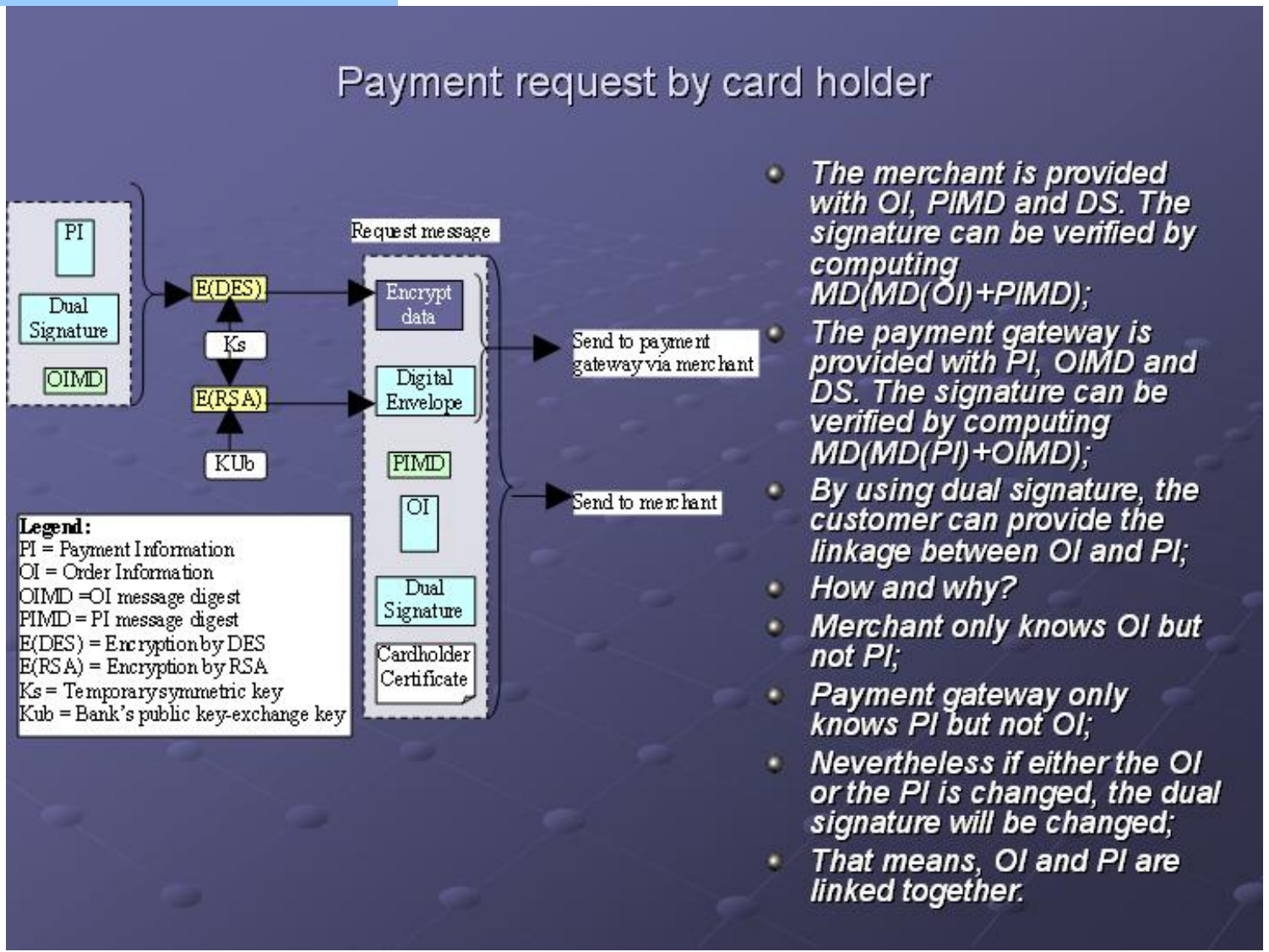
Dual signature *is an innovative method for resolving the following problem:*

- *The customer needs to send the order information (OI) to the merchant and the payment information (PI) to the bank;*
- *Ideally, the customer does not want the bank to know the OI and the merchant to know PI;*
- *However, PI and OI must be linked to resolve disputes if necessary (e.g., the customer can prove that the order has been paid);*

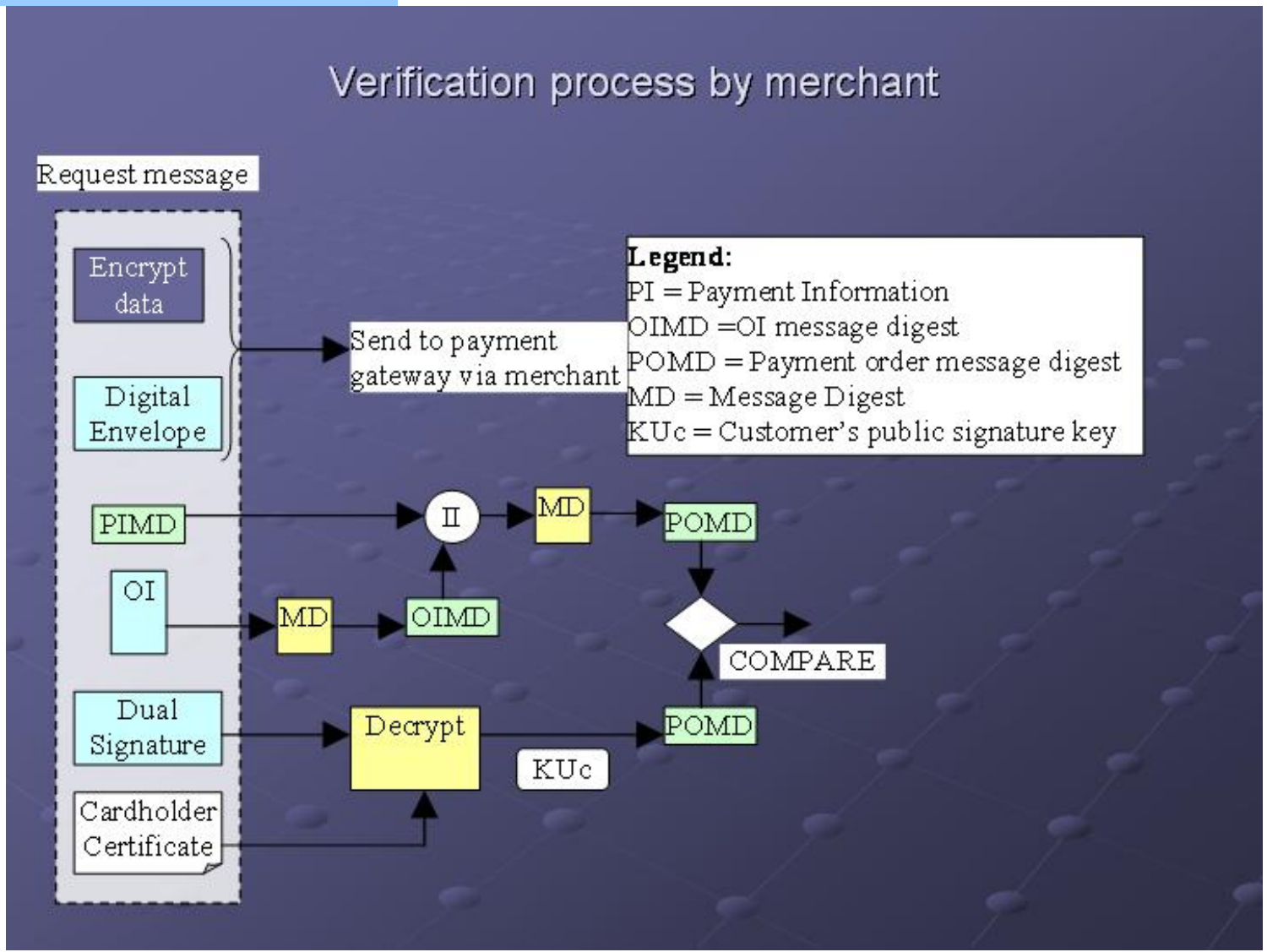


Legend:	PIMD = Payment Information message digest
PI = Payment Information	OIMD = Order Information message digest
OI = Order Information	POMD = Payment Order message digest
MD = Message digest function	E = Encryption (RSA)
II = Concatenation	Priv_key = Customer's private key for RSA

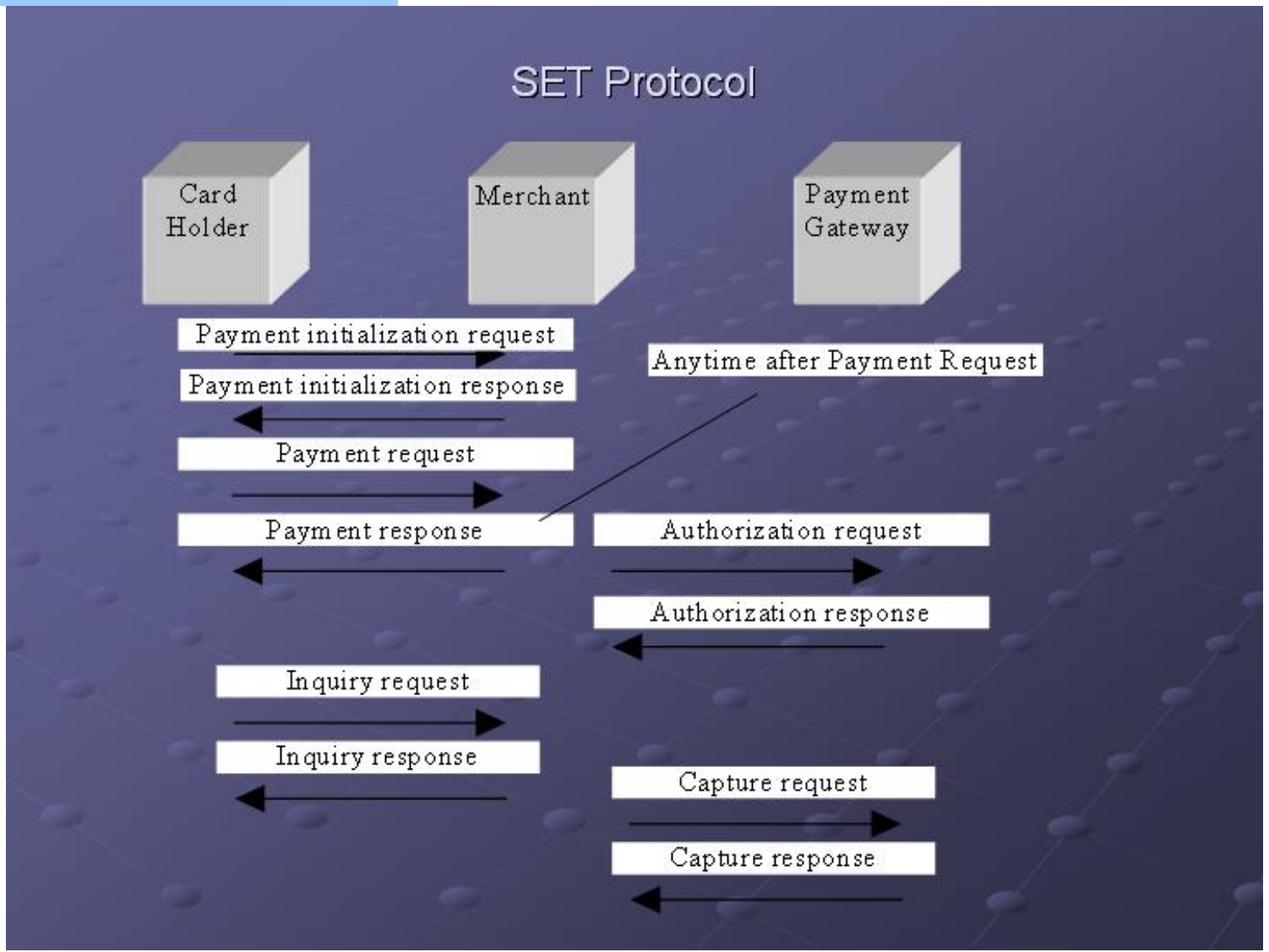
Java Card for SET App



Java Card for SET App



Java Card for SET App



Assignments Info

Mandatory conditions for the assignments:

1. The deadline is:
   date: 31.05.YYYY, hour: 23:50

2. The recipient for the assignment is ISM SAKAI platform:
   Assignments Menu

   The uploaded PDF file with the assignment is archived into a ZIP file with the following name rule:

   [LASTNAME][FIRSTNAME][ECPSSYYYY][ASSIGNMENTx].zip

   Where x=1…number of the projects within assignments list

3. The percentage of the assignments in final mark is **40%**

4. Paper format is the one provided by *www.secitc.eu*

Questions & Conclusions

Contact:

Cristian Valeriu TOMA
E-mail: cristian.toma@ie.ase.ro

E-mail for all the problems regarding ISM – IT&C Security Master:
ism.ase.ro@gmail.com
ism@ase.ro

Cristian Toma
IT&C Security Master

Dorobantilor Ave., No. 15-17
010572 Bucharest - Romania
http://ism.ase.ro
cristian.toma@ie.ase.ro
T +40 21 319 19 00 - 310
F +40 21 319 19 00