Java Card: a language for programming smart cards

THOMAS JENSEN

Projet Lande
IRISA/CNRS, Rennes

Overview of talk

1. Smart cards
2. The Java Card language and API
3. Java Card virtual machine and the CAP format

Smart cards

- Credit card sized plastic card
- Embedded memory and microprocessor chip
- with standardized I/O
- Contact or contact-less

Applications

Transportation

Access control (buildings, computers)

Banking (electronic wallet)

Satellite TV decoders, pay-per-view

Loyalty programs (air miles)

Advantages with smart cards:

- Security: Processing done on the card. Data (e.g. a private key) never leave the card.
- Portability: bring your own environment.

Physical characteristics

Processor

8 bit

3-4 MHz

24K ROM

16K EEPROM

512b RAM

Price: 3-6 €

Standards and specifications

Size, physical layout ISO 7816

Domain-specific EMV, GSM, IATA

Manufacturers Bull (F), Gemplus (F), Schlumberger (F), De La Rue (GB), IBM (US), Motorola (US), Giesecke & Devrient (D), Orga (D), . . .

Java Card

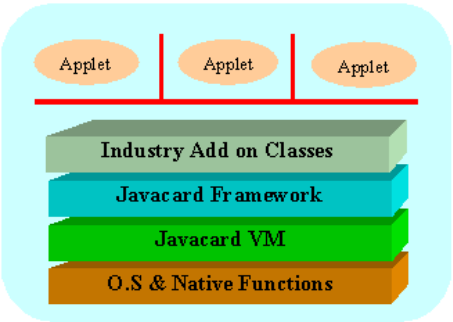
A high-level language for programming multiple-application smart card.

A cut-down version of Java with extra facilities for transactions and security.

Latest version: Java Card 2.1

Designed by Sun in connection with Java Card Forum.

Architecture of a Java Card



Java

Object-oriented language with

- classes,
- packages,
- interfaces,
- sub-classing and inheritance.

Example

```
package P1;

public class A {
private int value;
public int foo () {...}
public int bar () {...}
}

package P2;

class B extends A {
public int bar () {...}
}
```

Visibility modifiers

Reduce access to data and methods.

- `private`: visible only within class of declaration
- ⟨`package`⟩: visible within package
- `protected`: visible in package and sub-classes
- `public`: visible everywhere

Example

```
package P1;
public class A {
private int value;
int foo () {...}    // package vis.
public int bar () {...}}

package P2;
class B extends A {
A a;
public int bar () {... a.foo() ...}}
```


Java Card

Missing wrt. Java:

- No threads
- No object cloning, no garbage collection
- No “big” data types (`long`,`float`), no characters, `int` is optional
- No dynamic class loading

Added features:

- Atomic transactions
- Transient and persistent objects
- Applet firewall
- Interfaces for sharing objects

Electronic wallet example

```
package bank.purse;

import javacard.framework.*;

public class Wallet extends Applet {

   /* constants declaration */

final static byte Wallet_CLA =(byte)0xB0;
final static byte Deposit = (byte) 0x10;
final static byte Debit = (byte) 0x20;
final static byte Balance = (byte) 0x30;
final static byte Validate = (byte) 0x40;
final static byte PinTryLimit =(byte)0x03;
final static byte MaxPinSize =(byte)0x04;

OwnerPIN pin;
byte balance;
byte buffer[];
```


Java Card API 2.1

`java.lang`
: objects, exceptions

`javacard.framework`
: ISO constants, PIN codes, `Shareable`

`javacard.security` and `javacard.crypto`
: DES and RSA keys, digital signatures

```
private void debit(APDU apdu) {

 if ( ! pin.isValidated() )
   ISOException.throwIt(ISO.SW_PIN_REQUIRED);
 byte numBytes = (byte)(buffer[ISO.OFFSET_LC]);
 byte byteRead = (byte)(apdu.setIncomingAndReceive()
 if (byteRead != 1)
   ISOException.throwIt(ISO.SW_WRONG_LENGTH);
 if ( ( balance - buffer[ISO.OFFSET_CDATA]) < 0 )
   ISOException.throwIt(SW_NEGATIVE_BALANCE);
 balance = (byte)
 (balance - buffer[ISO.OFFSET_CDATA]);

 } // end of debit method
```


The life cycle of a smart card

Masking

Initialisation

Personalization

Execution of applets:

- installation: `install(APDU apdu)`
- selection: `select()`
- processing: `process(APDU apdu)`
- deselection: `deselect()`

Electronic wallet

```
private Wallet() {

  pin = new OwnerPIN(PinTryLimit,MaxPinSize);
  balance = 0;
  register();

} // end of the constructor


public static void install(APDU apdu){

  new Wallet();

} // end of install method

  public boolean select() {

  pin.reset();
  return true;

 }// end of select method
```


Persistent and transient data

Data is by default preserved between card sessions.

Transient objects have their contents erased at specified events (applet context change or card reset).

Atomic transactions

Methods

`JCSystem.beginTransaction` and
`JCSystem.commitTransaction`

delimit a transaction.

Updates are made to a transaction buffer.

All updates are committed atomically.

Applet firewall

Object owned by the applet that created it.

Applets are separated by a firewall

Context	=	applets from same package
	|	System

Context switch at method invocation.

A context stack keeps track of the current context

Object O owned by applet A1 can be accessed by applet A2 only if

$$context(A1) = context(A2).$$

System can access any object.

Passing through the firewall

Global arrays and entry point objects

$$User \longrightarrow System$$

Arrays can be global: accessible by all.

Objects in the system can be Entry Point Objects: methods callable from all contexts.

Ex: The `apdu` buffer.

Shareable interfaces

$$User \longrightarrow User$$

Shareable interfaces extend
`javacard.framework.Shareable`

Shareable interfaces specify a set of methods made available to other contexts.

Inter-context object sharing

SI extends `javacard.framework.Shareable`

P1.A	P2.B
class C implements SI	
construct C object O	
	declare object ref. R with interface SI
	request shared object from P1.A
cast O to `Shareable`	
send O to P2.B	
	cast O to SI

The Java Card Virtual Machine

"Aside from its persistent nature, the Java Card VM is just like the Java VM"

Java Card Runtime Environment 2.1 Specification

However,

- Unsupported bytecodes
  `fload`, `lstore`, `monitorenter`, ...
- Managing applet firewall
- Compressed package representation (CAP format)
  Specialised bytecode formats
  Method table compression

Converted APplet format

CAP file format: classes and Interfaces of a Java package.

CAP components

- **Header** Java Card name of package, linkable?, contains applets?,...
- **Directory** Size of all components.
- **Applet** # Applets in package, their applet id and offset of the `install` method.
- **Constant Pool** Merged constant pools
- **Class** All classes in package
- **Method** All methods (code, exception,...).

And also

- **Referenced Packages**
- **Static Fields**
- **Reference Location**
- **Export**
- **Descriptor**

From strings to tokens

Class files use Unicode strings for references

CAP file: Opaque tokens (numbers)

Tokens $\neq$ offsets

Tokens for

- packages
- classes
- fields
- virtual methods

Export file contains tokens for all externally visible items.

Constant pool - class files

Code:

$\vdots$

`putfield` i

$\vdots$

Constant pool:

$j \mapsto$ `fr.irisa/`$\ldots$`/C.class`

$\vdots$

$i \mapsto (class\_index, name\_and\_type\_index)$

$\vdots$

$k \mapsto (l, m)$

$\vdots$

$l \mapsto$ `foo`

$m \mapsto$ `I`

Constant pool - CAP format

The class file CP entry for an instance field

$$n \mapsto \begin{cases} class\_index, \\ name\_and\_type\_index \end{cases}$$

is transformed into

$$n \mapsto \begin{cases} class\_ref, \\ field\ token \end{cases}$$

where a class reference has form

$$class\_ref = \begin{cases} offset\ into\ Class\ component, \\ (package\ token, class\ token) \end{cases}$$

Specialized byte codes: `putfield`

Information moved from constant pool to byte codes.

4 different versions:

`putfield_`{a,b,s,i}

- `a` : reference
- `b` : byte or boolean
- `s` : short
- `i` : integer

In addition, for each type $\langle t \rangle$,

`putfield_`$\langle t \rangle$_{$\langle\ \rangle$,w,this}

- $\langle\ \rangle$: normal (one-byte) index
- `w` : wide (two-byte) index
- `this` : objectref to `this` in local variable 0

Method tokens and overriding

```
class A
   foo () { ... }
   bar () { ... }
```

Method table for objects of class A.
$\langle A.foo \rangle$ = pointer to code for `A`'s `foo` method

`foo`	$\langle A.foo \rangle$
`bar`	$\langle A.bar \rangle$

```
class B extends A
   foo () { ... }
```

Method table for objects of class B:

`foo`	$\langle B.foo \rangle$
`bar`	$\langle A.bar \rangle$

Method tables and tokens

In CAP: use tokens as indices into method tables.

Method table for class `B`.

1	$\langle B.foo \rangle$
2	$\langle A.bar \rangle$

Tokenization constraint:

> a method overriding another method must receive the same token as the overridden method.

Method table compression

`P1.A`

$\texttt{f}_1()\quad \{\ldots\}$

$\vdots$

$\texttt{f}_{17}()\quad \{\ldots\}$

Table for `P1.A`

1	$\langle \texttt{P1.A.f}_1 \rangle$
$\vdots$	$\vdots$
17	$\langle \texttt{P1.A.f}_{17} \rangle$

`P2.B extends P1.A`

$\texttt{g}()\quad \{\ldots\}$

Table for `P2.B`

1	$\perp$
$\vdots$	$\vdots$
17	$\perp$
18	$\langle \texttt{P2.B.g} \rangle$

Method table compression II

Solution: Split into two token spaces and method tables

- one for package-visible methods
- one for protected/public methods

"Private" tokens: start at 0 when subclass is in a different package

"Public" tokens" start at 1 + max. token in superclass

Restriction: Cannot override package-visible method and change its visibility modifier.

Method table compression III

`P1.A`

`public` $\texttt{f}_1()\quad \{\ldots\}$

$\vdots$

`public` $\texttt{f}_{17}()\quad \{\ldots\}$

Table for `P1.A`

1	$\langle \texttt{P1.A.f}_1 \rangle$
$\vdots$	$\vdots$
17	$\langle \texttt{P1.A.f}_{17} \rangle$

`P2.B extends P1.A`

`public` $\texttt{g}()\quad \{\ldots\}$

Table for `P2.B`

1	$\langle \texttt{P1.A.f}_1 \rangle$
$\vdots$	$\vdots$
17	$\langle \texttt{P1.A.f}_{17} \rangle$
18	$\langle \texttt{P2.B.g} \rangle$

Method table compression IV

CAP format permits

Base

18

Table for `P2.B`

18	$\langle \texttt{P2.B.g} \rangle$

but also

Base

16

Table for `P2.B`

16	$\langle \texttt{P1.A.f}_{16} \rangle$
17	$\langle \texttt{P1.A.f}_{17} \rangle$
18	$\langle \texttt{P2.B.g} \rangle$

Mixing method tables and dynamic method search.

Conclusions

Java Card: high-level, multi-application software development to smart cards.

An application area for formal techniques:

- Optimize the use of the limited card resources (memory in particular)
- Certify implementations and applications with respect to international security standards