Java Card: Programming Guidelines and Best Practise

Introduction

The purpose of this document is to provide proven technical guidelines to Java Card developers,

focusing on applet design and optimization. We hope these guidelines will help face the specificity of Java Card development,

i.e.:

• Card resources are extremely constrained.

EEPROM size typically is 32-64Kb and is used to hold post-issuance applets as well as application data.

RAM size is usually around 4-8Kb, and must accommodate for application stack, transient data, and various runtime buffers.

You just can't afford any waste.

• CPU power is very limited. Also, Java Card applets are much slower than native code,

so you don't want to perform unnecessary operations.

• Even though your applet runs fine on your platform, you still want to save as many resources as possible,

so that it will also run on more limited platforms (less stack, smaller transaction buffer, etc). WORA, remember?

Write Once, Run Anywhere

Applet Design

Java Card is not Java. Fancy object-oriented designs lead to fat and slow applets, because of:

• class/interface overhead in the CAP file

• virtual method overhead at run time

As always, there is certainly room for improvement,

but following them will surely prevent a large number of common (and not-so-common) mistakes.

High-level design

It’s usually at design time that tragic mistakes are made:

if this happens, nothing short of a complete re-design will save you… To avoid this,

please read this section very carefully and make sure you apply these guidelines.

Start with a reasonably object-oriented design. This will help you understand how the applet works,

especially if it has complex features (PKCS, EMV, etc).

Breaking down the problem into smaller objects is still the best way to figure things out.

Still, you must avoid the following pitfalls:

• class proliferation: everything is not an object,

so please keep the number of classes / abstract classes/interfaces under control

• utility classes/interfaces: instead of defining classes containing only static methods,

move those methods to classes which are actually instantiated.

The same goes for any interface holding only constants: move the constants to actual classes

• Deep class hierarchies: the deeper the class tree, the costlier the invocation of inherited methods gets.

Inheritance also means that you will have nested constructor calls, and this puts the stack at risk.

In short, the necessity of inheritance must be crystal-clear: if not, it must go.

• design patterns: patterns look great in specs, but their implementation means lots of abstract classes,

interfaces and deep class trees

• systematic get()/set () methods: again, this looks great in UML charts,

but it's mostly dead weight for Java Card applications.

Increasing member visibility will allow you to remove them, hence reducing code size and improving the execution speed.

• package proliferation:

there's really no need to break an application into packages unless one of the packages is going to be used by another applet.

It may look nicer, but cross-package method invocation has a cost: avoid it if it's not mandatory.

• C-style error handling: do not use C-style error handling in your methods,

i.e. check the return code to see if an error has occurred.

This produces inefficient - not to mention ugly - code,

because you keep running error-related code even if no error has occurred.

Your design must use Java exceptions instead: errors only need to be handled once they have occurred!

Once the design is complete, your application should include one or two packages,

one to ten classes and not more than three levels of classes.

Anything larger needs to be thoroughly investigated and justified.

Once the applet is implemented and tested,

you can be confident that functional requirements have been correctly understood and implemented.

At this stage, you may start breaking the OO design and optimizing to decrease code size and/or improve performance.

Data storage

Your design must take these constraints into account:

• writing in RAM is fast

• writing in EEPROM is awfully slow

• allocating objects is even worse

• garbage collection is not part of Java Card 2.1.1

Memory allocation

The key rule is to create all instance data at installation time,

i.e. in the applet constructor called by **javacard.framework.Applet.install():** object construction using new,

transient array allocation, all explicit initializations.

This way, there's no risk to face a memory shortage during the life of the applet:

everything is already reserved at installation time. Furthermore, applet commands won't be slowed down by memory allocation.

Memory writes

Objects are stored in EEPROM. This means that every time you're doing "myObject.myByte = (byte)5", you are writing in EEPROM.

This must be avoided as much as possible. Obviously, you need to store persistent data,

but there are many cases when you only need to store temporary data

(intermediate result, data which is sent back to the terminal, etc).

This is where the APDU buffer comes in handy! After all, it's a "normal" byte array, so use it as much as you can.

Security

If the applet is submitted to a security evaluation (Common Criteria, etc.),

some specific steps may be taken: • isolate all security functions (key/PIN handling, etc.) in separate class.

This way, only these classes need to be documented and evaluated.

• use the Java Card API as much as you can (PIN, keys, etc).

If the platform offers the security services you need, it's useless and unsafe for the applet to implement them.

Fixed size PIN

Use fixed-size PINs, i.e. store the PIN length in the PIN itself and pad the remaining bytes.

This helps prevent timing attacks and also simplifies PIN handling.

Anti-DFA RSA signatures

Verify all RSA signatures before sending the result to the terminal.

This will slow down the signature operation, but it will prevent DFA attacks.

Miscellaneous

Keep an eye on platform bugs: they may have an impact on your design.

If the applet must be interoperable, do NOT use any proprietary APIs.

Applet Optimization

This section lists a number of real-life (i.e. tested and proven) optimizations.

Before applying them, please keep in mind the golden rules of optimization:

• focus on the most frequently run code: this is where you will make a difference

• you usually have to choose between size and speed

• run your benchmarks after each optimization pass

• run your validation tests after each optimization pass

Reducing code size

Avoid complex object-oriented designs

If you read the previous section, this should be obvious. In short:

• Keep the number of classes/interfaces to a minimum.

Combine “similar” classes and discriminate them at instantiation time using constructor parameters

• Keep the number of methods to a minimum. In particular, get rid all get()/set() methods :

increase the visibility of data members (e.g. from private to package visible) to reduce the number of get()/set() methods.

Since you save method calls, this will also improve performance

Tradeoff: encapsulation is weakened. If you control all classes in your package, this shouldn’t be an issue.

Remove dead code

Hunt down unused variables and code. Sounds obvious, but you’d be surprised...

Also, try to keep the initialization / personalization as short as possible.

It’s only going to be run once, so it’s pretty much dead weight.

Maybe the more complex operations can be performed by the personalization system?

Factor duplicated code

Hunt down all redundant code and factor it.If this means breaking down classes and replacing them with private/static methods,

do it.

Tradeoff: possible slowdown if the code is called very often.

Limit number of parameters

Try to use no more than 3 parameters for virtual methods and 4 for static methods.

This way, the compiler will use a number or bytecode shortcuts,

which help reduce code size: aload_x (1 byte) instead of aload x (2 bytes), etc.

Tradeoff: noticeable slowdown if this means saving/reading data from object instances.

Reducing EEPROM consumption

Recycle all objects.

The Java Card standard doesn't require garbage collection.

So unless you insist on wasting memory, you have to keep track your old objects and reuse them.

Remember:

the Java Card Virtual Machine runs "forever",

so if an object becomes unreachable, its memory is gone "forever" (or at least until you delete the applet).

Even if your platform provides proprietary garbage collection, you'd better reuse your objects.

Allocating new objects is slow, and so is garbage collection. You get the point...

Allocate arrays carefully.

The OS allocates memory in 32-bytes chunks, called clusters.

A cluster cannot be shared between two objects, so any object will always eat at least one cluster,

no matter how small it is. Furthermore, the Virtual Machine appends a header to all objects:

• 6 bytes for “normal” objects

• 8 bytes for primitive type arrays

• 12 bytes for object arrays

This means that instead of creating several small arrays of the same type,

you should combine them into a single array, accessing the latter using fixed offsets.

Tradeoff: slight increase of code complexity.

Example:

```
// Multiple arrays
class TestApplet extends Applet {
  private byte[] array1;
  private byte[] array2;
  private byte[] array3;
  private byte[] array4;
  private byte[] array5;
  private byte[] array6;
  ...
  public void process(APDU apdu) {
    array1 = new byte[(byte)8]; //1 cluster
```

```
    array2 = new byte[(byte)12]; //1 cluster
    array3 = new byte[(byte)23]; //1 cluster
    array4 = new byte[(byte)24]; //2 clusters
    array5 = new byte[(byte)25]; //2 clusters
    array6 = new byte[(byte)32]; //2 clusters
  }
  ...
}
```

To store a total of 124 bytes, this code requires 9 32-byte clusters:

this means that it actually reserves 288 bytes of EEPROM!

```
// Single array
class TestApplet extends Applet {
  private byte[] array7;
  ...
  public void process(APDU apdu) {
    ...
    array7 = new byte[(byte)124];
    //5 clusters
    ...
  }
}
```

To store the same 124 bytes, this code requires only five 32-byte clusters:

this means that it actually reserves “only” 160 bytes of EEPROM.

Declare static members

Whenever possible, declare primitive fields as static. This will save memory.

Example:

```
byte myByte = (byte)1;
// 2 bytes of EEPROM
static byte myByte = (byte)1;
// 1 byte of EEPROM
```

Reducing RAM consumption

Reuse local variables

Instead of allocating a new local variable any time you need one, try to reuse one that has been previously declared.

Tradeoff: abusing this technique produces unreadable code.

Allocate transient arrays carefully

Instead of creating several small transient arrays of the same type,

combine them into a single array and access it using fixed offsets.

Tradeoff: slight increase of code complexity.

Limit number of parameters

Try to use no more than 3 parameters for virtual methods and 4 for static methods.

This way, the compiler will use short bytecode instructions,

which help reduce code size: aload_x (1 byte) instead of aload x (2 bytes), etc.

Tradeoff: noticeable slowdown if this means saving/reading data from object instances.

Avoid deeply nested method calls

Dangerous nested calls usually happen with deep class trees (base class method calls) and with recursion.

The latter yields fancy and compact code, but it also tends to smash the stack very quickly… Triple-check the end

condition(s) and make sure the tests actually trigger the worst case.

Tradeoff: possible code size increase.

Beware of local variables in switch/case structures

Instead of allocating a new local variable in each case, declare only one before switch.

Example:

```
switch (myValue) {
  case VALUE1:
    short s1;
    ...
    break;
  case VALUE2:
    short s2;
    ...
    break;
  case VALUE3:
    short s3;
    ...
    break;
}
```

And improved:

```
short s;
switch (myValue) {
  case VALUE1:
    ...
    break;
  case VALUE2:
```

```
    ...
    break;
  case VALUE3:
    ...
    break;
}
```

Improving execution speed

Switch on compiler optimization

If your compiler provides optimization flags, use them (-O for javac).

Tradeoff: probable code size increase.

Avoid complex object-oriented designs

See Applet design and Applet Optimization for details.

Tradeoff: encapsulation is weakened. If you control all classes in your package, this shouldn't be an issue.

Favor static, private, & final methods

All public/protected Java methods are implicitly virtual methods (unlike in C++, where they had to be declared virtual).

This means that dynamic binding always takes place,

i.e. the Virtual Machine must always determine the actual type of an object before invoking a public/protected method.

This lookup is costly, especially if the method being invoked is inherited from a base class.

This is the main reason why deep class trees are not welcome.

To handle this problem:

1. favor static methods, because they're not subjected to dynamic binding, which makes them faster to invoke

2. if a method can't be static, try to make it private. Private methods cannot be overridden in derived classes,

so it's easier for the Virtual Machine to locate them

3. if a method can’t be private, make it final as soon as possible. This will help the Virtual Machine

Use native APIs

Whenever the platform provides native code to perform an operation, use it!

Native methods, are way faster than any clever Java code you could come up with.

In particular:

• use Util.arrayFillNonAtomic(),Util.arrayCopy(),Util.arrayCopyNonAtomic() to initialize or modify arrays

• Util.arrayFillNonAtomic() is particularly useful to pad buffers during cryptographic operations

• use Util.getShort(), Util.setShort() to handle byte[]/short conversions

Avoid unnecessary initializations

The Java Card Virtual Machine guarantees that newly-allocated data members and local variables are set to 0/false/null.

Don’t use exceptions for flow control

Using exceptions for flow control is never a great idea, and Java Card is no exception.

Exception handling is terribly slow and shouldn’t be used for anything else than exception handling.

In other words, if you were thinking about using throw to simulate goto, just think harder and find a better solution.

Use RAMbuffers

Writing in EEPROM is about 1,000 times slower than writing in RAM, so the less you do it, the better.

You can work with the APDU buffer or with transient arrays to store session data and temporary results.

This works particularly well for intermediate cryptographic operations and it’s much safer as well.

Clean up your loops

Accessing an instance member is costlier than accessing a local variable. If you have to perform repeated accesses,

store the instance member in a local variable on first access and then use the local variable only.

The same goes for array accesses (array[i], array.length) and method calls. This will yield a large performance increase.

Example:

```
// bad
for(short i=0; i< myArray.length; i++){...}
// good
short s = myArray.length;
for(s i=0; i< s; i++){…}

// bad
for(byte i=0; i< myObj.size(); i++){...}
// good
byte b = myObj.size();
for(byte i=0; i< b; i++){...}
```

Use the evaluation order of logical expressions

Logical expressions (tests, etc) are evaluated from left to right. If a condition fails, the rest of the test is not evaluated.

If one of the conditions is almost always false, check it first : you'll avoid executing unnecessary code.

Use return values as often as possible

The return value of many APIs is meaningful: using it will save unnecessary operations.

Example:

```
// bad
short offset = util.setShort(bArray, (short)0, sValue);
offset = offset + 2;
Util.arrayCopy(src, srcOff, dest, destOff, length);
offset = offset + length;
```

```
// good
offset = Util.setShort(bArray,(short)0, sValue);
offset = Util.arrayCopy(src, srcOff, dest, destOff, length);
```

Use transactions with care

Since many native APIs already transaction write operations,

make sure that any explicit use of transactions in the applet is mandatory.

In particular, do not use Util.arrayCopy() if you don't need transactions : use Util.arrayCopyNonAtomic() instead.

Check class & instruction at the same time

This works particularly well when OP Secure Messaging is used, because all secure commands will use two different class values.

Example:

```
short cla_ins = Util.Setshort(buffer, IOS7816.OFFSET_CLA);
switch(cla_ins){
  case READ_BIN_CLA_80 :
  case READ_BIN_CLA_84 :
    ...
    break;
  case VERIF_PIN :
    ...
    break;
  ...
  default: if ( ((cla_ins & 0xFF00) == 0x8000) ||
    ((cla_ins & 0xFF00) == 0x00) ||
    ((cla_ins & 0xFF00) == 0x8400)
    // exception INS_NOT_SUPPORTED;
  else
    // exception CLA_NOT_SUPPORTED;
```

Check P1 & P2 at the same time

Whenever possible, check P1 and P2 at the same time. Your code will be both faster and more compact. Example:

```
 switch (Util.getShort(buffer, ISO7816.OFFSET_P1)) {
  case (short)0x41B6:
   MseSetDigitalSignatureTemplate(...);
   break;
  case (short)0xC1B8:
   MseSetConfidentialityTemplate(...);
   break;
  case (short)0xC1B4:
   MseSetCryptographicChecksumTemplate(...);
   break;
  default:
   ISOException.throwIt(ISO7816.SW_INCORRECT_P1P2);
   break;
 }
```