Java Card

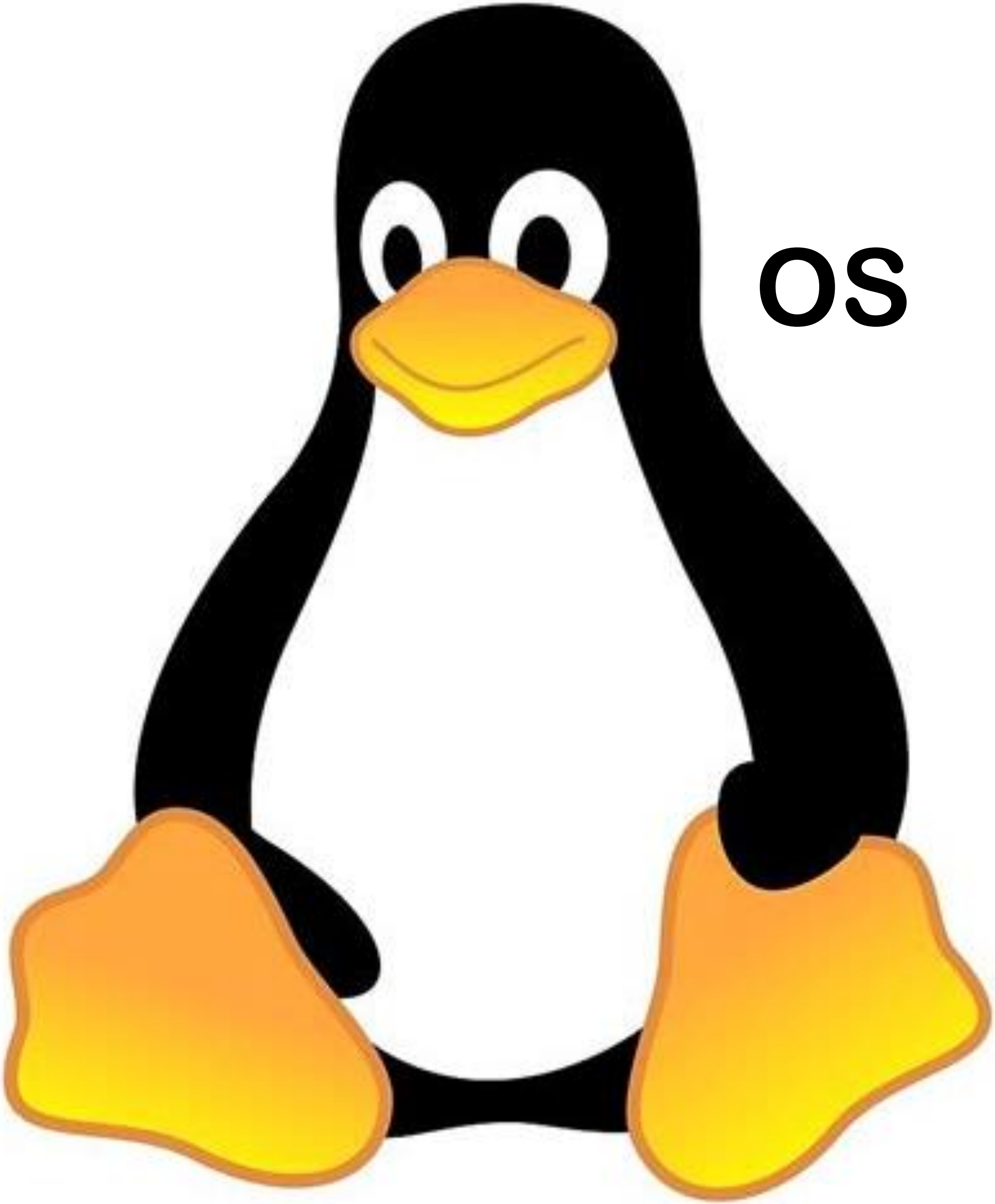
Erik Poll
Digital Security
Radboud University Nijmegen

Radboud University Nijmegen

Contents

- Smartcard Operating Systems
- Java Card architecture
- Java vs Java Card
  - APDUs
  - transient and persistent data
  - transactions
  - crypto

- Fun with transactions



OS

smartcard OS

Smartcard OS

No management of multiple processes, user accounts, loads of device drivers, etc. etc. as in normal modern OSs

Tasks:

- Life-cycle management of card
- Application management
- Memory management
- Some libraries for I/O & crypto
  - just one simple device driver for I/O using ISO7816 APDUs
- Hardware error handling

Smartcard OS evolution

1. No OS: one application, burnt into ROM
2. Standard libraries in ROM, applications in EEPROM
3. Proprietary operating systems
   - programmed in machine code with proprietary instruction set
   - often providing ISO7816-4 file system
4. Modern multi-application smartcards
   - MULTOS
   - JavaCard
   - Windows for smartcards †

Smartcard life cycle (ISO 10202-1 - cancelled)

1. Production of chip & card
    - testing & removing test functionality
2. Card preparation
    - putting it in plastic card & ‘completing’ the OS
3. Application preparation incl. personalisation
    - initialising applications
    - personalisation - both electrically & optically
4. Card utilisation
    - (de)activation of applications
5. End of card utilisation
    - de-activating applications and/or card

OS completion

1. Initially, card contains ROM mask
2. Simple loader in ROM executed to load EEPROM
3. Checksum computed
4. Switch to mode where code in ROM and EEPROM can be executed

Typical application life cycle

1. Installation of application (aka applet)
   - uploading & installing code
2. Personalisation
   - uploading application data
   - afterwards, application starts in normal active life
3. End-of-life
   - disabling all functionality
   - possibly leaving logging functionality enabled
   - switch to end-of-life state by external command or when card notices something suspicious

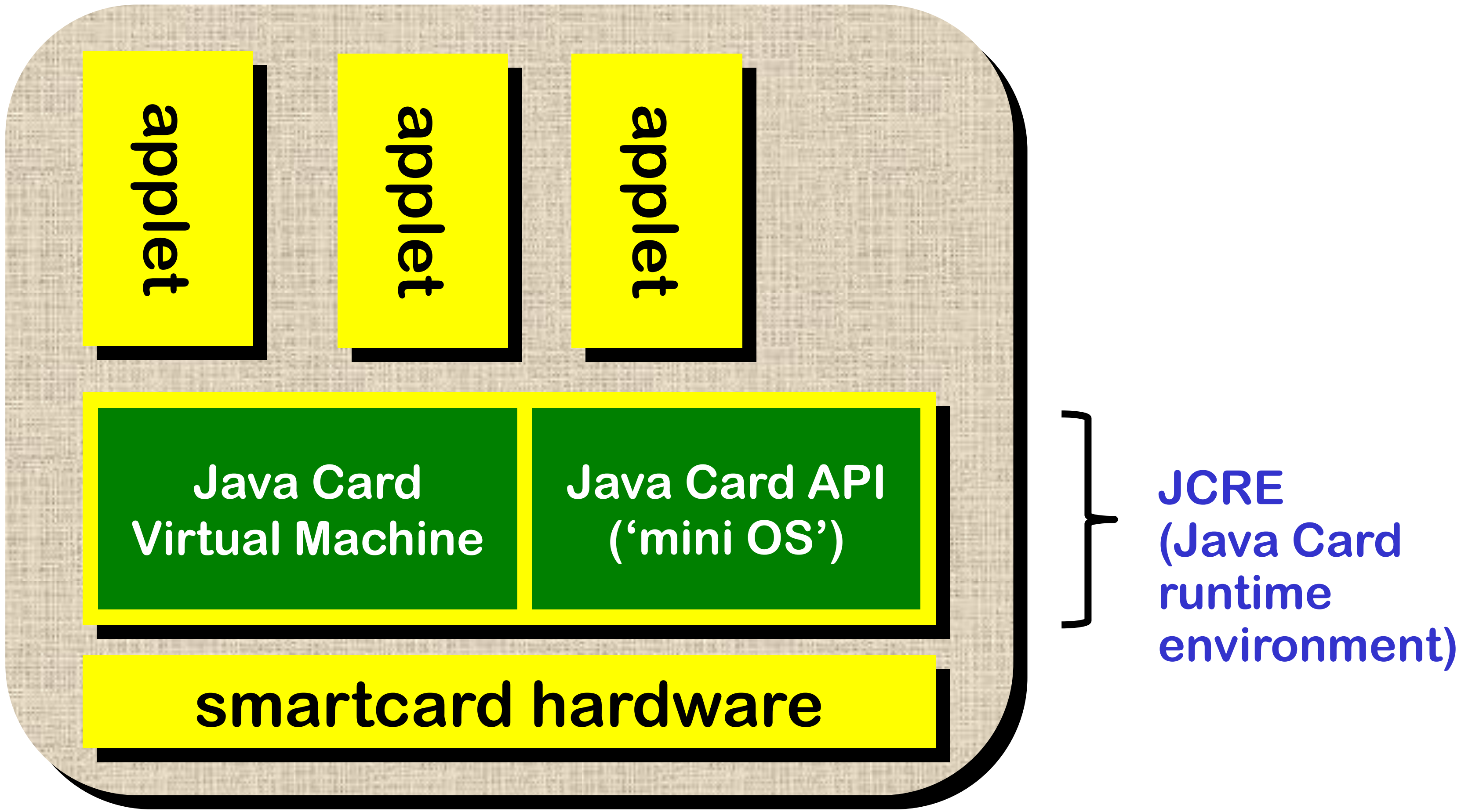
Traditional smartcard vs JavaCard (or MULTOS)

- One program
- Written in machine-code for a specific chip
- Burnt into ROM or uploaded once to EEPROM

- Programs (applets) written in high-level
- Compiled into bytecode
- Stored in EEPROM
- Interpreted on card

- Multi-application: several applets on one card
- Post-issuance: adding or deleting applets on cards "in the field"

Java Card architecture



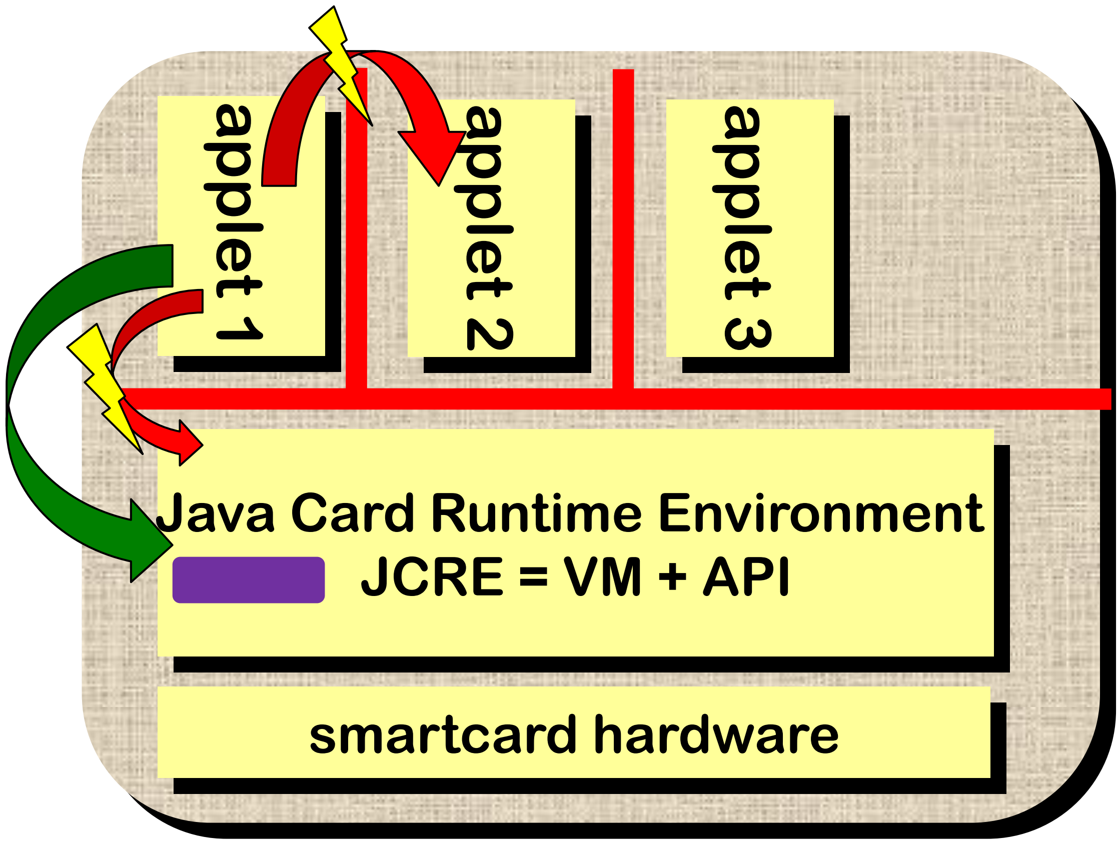
Java Card vs Java architecture

Java Card applets are executed in a sandbox,
like Java applets on a Java VM

But important differences with the Java sandbox model :

- No bytecode verifier on most cards (due to memory required)
- Downloading applets is controlled by digital signatures using the Global Platform API (formerly Visa Open Platform)
- Sandbox more restrictive, and includes runtime firewall to provide protection between applets and between applets and the OS
  - in particular: applets cannot share references

Java Card firewall



RUNTIME checks to prevent access to

- fields & methods of other applets
- objects owned by other applets

even if these are `public`!

Exceptions:

- applets can access some JCRE-owned objects, eg APDU buffer

There is a way for applets to expose functionality to each other, using `Shareable` interfaces but you won't be needing that

Advantages of JavaCard?

- Vendor-independence
- Easy to program
  - higher-level language => smaller programs with fewer bugs
  - standard functionality (eg for PINs) provided once by the API
    - esp security-sensitive functionality, so people don't mess things up by implementing this themselves
- Open standard
  - no reliance on security-by-obscurity
    - specs can be studied and criticised
  - BUT: all implementations tightly closed, and bug-fixes in the specs happen secretly

Disadvantages of JavaCard?

- **Overhead of VM**
  makes cards slow and requires lots of memory => expensive
- **Trust?**
  how secure is the whole JavaCard infrastructure?
  complicated platform, and complexity <≠> security
- **Ease of programming?**
  may be deceptive, and invite non-experts to program cards and make silly mistakes
- **Easy for attackers to experiment with cards?**
  esp. if they can get blank JavaCard of the some brand to look for bugs or weaknesses. NB security by obscurity has its merits!

Does simpler application code using a standard API outweigh having to trust a bigger platform ?
It is hard to compare the TCBs for JavaCard vs proprietary OS

The Java Card language

- A dialect of Java tailored to smartcard
  - superset of a subset of Java
- Subset of Java (due to hardware constraints)
  - no threads, `doubles`, `Strings`, multi-dimensional arrays, and a *very* restricted API
  - support for `int` optional
  - garbage collection optional
- But… with some extras (due to hardware peculiarities)
  - communication via APDUs
  - persistent & transient data in EEPROM &RAM
  - transaction mechanism

Don't create garbage !

JavaCards usually have no garbage collector, and *very* limited memory!

Hence no calls of

```
new ...
```

anywhere **in your code** ***except*** **in the installation phase (e.g. the applet's constructor)**

More generally, *JavaCard programs should **not** look like normal Java programs, but more like C code written in Java syntax*

In particular, go easy on the OO: most objects should be byte arrays.

The Java Card language

- JavaCard uses an optimized form of class files, called cap-files

  1. compiler translates .java to .class
  2. converter translates .class to .cap
     compressing code, eg replacing method & field names by offsets

- JavaCard uses 16 bit arithmetic, not 32 bit

16 bit arithmetic

JavaCard code contains many `(short)` casts:

all intermediate results (which are of type `int`) must be cast to `short`

so that results are the same on a 16 bits JavaCard VM

as on a normal 32 or 64 bits Java VM

```
short s; byte b;
s = b+s+1;
     // not ok, compiler complains; there is an
     // implicit cast from 32 to 16 bit
s = (short)(b+s+1);
     // not ok, converter complains; a 16 bit CPU will
     // implicitly cast intermediate result to 16 bit
s = (short)(b+(short)(s+1)) // ok
```

Moral of the story: your JavaCard code should look really ugly, with `(short)` casts all over the place.

(un)signed bytes

- Bytes in Java and Java Card are signed
  ie. for any `byte b`
  `b` ∈ {-128, ..., 127}

- To interpret byte as unsigned , write
  `b & 0x00FF` ∈ {0, ..., 255 }

***Moral of the story:** your JavaCard code should look REALLY ugly, with `(short)` casts and `& 0x00FF` all over the place.*

the Java Card API

- A subset of Java's API
  - no need for most standard I/O classes
  - no `Strings`
  - no `clone()` in `Object`
  - `...`
- plus some extras for
  - smartcard I/O with APDUs using ISO7816
  - persistent and transient data
  - transactions

Java Card API packages

- `java.lang`

  `Object, Exception, ...`
- `javacard.framework`

  `ISO7816, APDU, Applet, OwnerPIN, JCSystem`
- `javacard.security`

  `KeyBuilder, RSAPrivateKey, CryptoException`
- `javacardx.crypto`

  `Cipher`

Card-terminal communication

Communication via APDUs, as defined in ISO7816,
using API class `javacard.framework.APDU`

1. JavaCard OS sends a command APDU to an applet, by invoking the

   ```
   process (APDU the_apdu)
   ```

   method of an applet.

   This APDU object `the_apdu` contains a byte array buffer

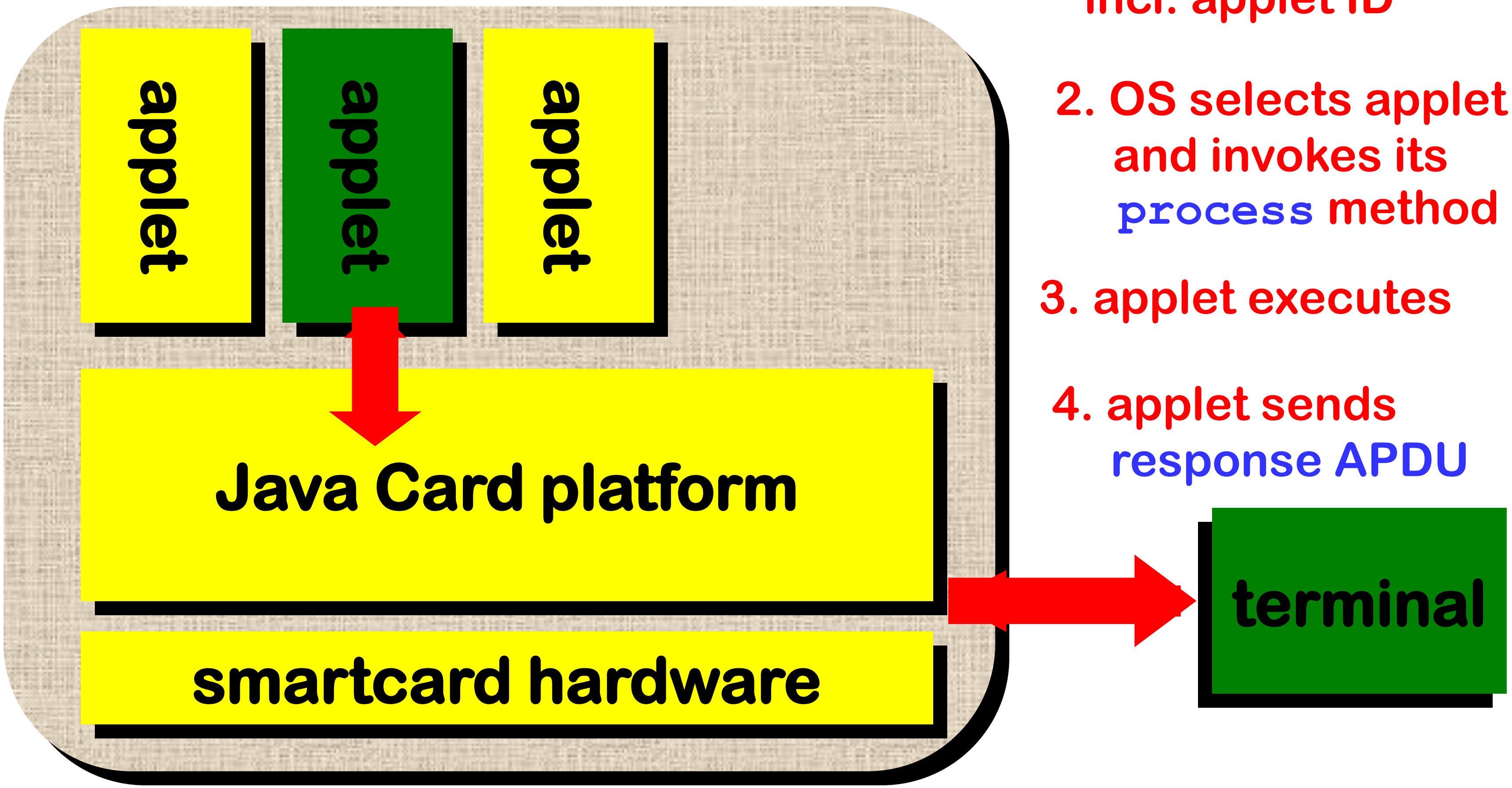
   ```
   the_apdu.getBuffer()
   ```

2. Card sends a response APDU back,
   by invoking API methods on this APDU object, eg

   ```
   the_apdu.sendBytes(offset,length)
   ```

See the `process` methods of the `*Applet.java` examples linked from Brightspace on how APDUs are used.

Java Card I/O with APDUs



1. terminal sends command APDU incl. applet ID
2. OS selects applet and invokes its `process` method
3. applet executes
4. applet sends response APDU

RMI

- Dealing with APDUs cumbersome
- JavaCard 2.2 introduced RMI (Remote Method Invocation)
  - Terminal invokes methods on applet on the smartcard, eg

```
int doPayment(short amount,
              boolean PIN_required,
              byte[] description)
```

  - Platform translates this method invocation into APDUs by (un)marshalling the parameters & return value

BUT: only works if method parameters fit in the APDU buffer

- So there is limit on total size on all method arguments combined.

Nobody in industry is using RMI; easier if you don't.

Contents

- Smart Card Operating Systems
- Java Card architecture
- Java vs Java Card
  - APDUs
  - transient and persistent data
  - transactions
  - crypto

- fun with transactions

Recall the smartcard hardware

- ROM
  - program code of VM, API, and pre-installed applets
    - though most of that will be in EEPROM
- EEPROM
  - program code of applets
  - persistent storage of the data
- RAM
  - transient storage of data

JavaCard programmer only has to worry about RAM and EEPROM

Data stored in EEPROM is persistent, and is kept when power is lost
data stored in RAM is transient, and is lost as soon as power is lost

Persistent (EEPROM) vs transient (RAM)

- Heap in EEPROM, stack in RAM
  - objects (incl. their fields) are stored in EEPROM
  - local variables and method arguments are in RAM

Only exception:

- API methods allow allocation of arrays in RAM instead of EEPROM

  NB only the *content* of these arrays in RAM;
  the *array header, incl. its length*, are in EEPROM

Persistent (EEPROM) vs transient (RAM)

```
public class MyApplet extends javacard.framework.Applet{
 byte[] p;
 short balance;
 SomeObject o;
 p = new byte[128];
 o = new SomeObject();
 balance = 500;

 public short someMethod(byte b) {
  short s;
  Object oo = new Someobject();
  ...
}
```

Persistent (EEPROM) vs transient (RAM)

```
public class MyApplet extends javacard.framework.Applet{
 byte[] p; // persistent, ie in EEPROM
 short balance;
 SomeObject o;
 p = new byte[128];
 o = new SomeObject();
 balance = 500;

 public short someMethod(byte b) {
  short s; // transient, ie in RAM
  Object oo = new Someobject();
  ...
}
```

Persistent (EEPROM) vs transient (RAM)

```
public class MyApplet extends javacard.framework.Applet{
 byte[] p; // persistent, ie in EEPROM
 short balance;
 SomeObject o;
 p = new byte[128];
 o = new SomeObject();
 balance = 500;

 public short someMethod(byte b) {
  short s; // transient, ie in RAM
  Object oo = new Someobject();
  ...
}
```

Spot the potential problem!

Persistent (EEPROM) vs transient (RAM)

```
public class MyApplet extends javacard.framework.Applet{
 byte[] p; // persistent, ie in EEPROM
 short balance;
 SomeObject o;
 p = new byte[128];
 o = new SomeObject();
 balance = 500;

 public short someMethod(byte b) {
  short s; // transient, ie in RAM
  Object oo = new Someobject();// potential garbage ??
  ...
}
```

Allocating arrays in RAM

```
public class MyApplet extends javacard.framework.Applet{
 byte[] t, p;
 short balance;
 SomeObject o;

 // persistent array p and persistent object o
 p = new byte[128];
 o = new SomeObject();
 // transient array t
 t = JCSystem.makeTransientByteArray((short)128,
                                    JCSystem.CLEAR_ON_RESET);
```

Allocating arrays in RAM

```
public class MyApplet extends javacard.framework.Applet{
 byte[] t, p;
 short balance;
 SomeObject o;
```

NB `t` is persistent,
`t.length` is persistent, and
only the contents `t[..]` is transient

```
 // persistent array p and persistent object o
 p = new byte[128];
 o = new SomeObject();
 // transient array t
 t = JCSystem.makeTransientByteArray((short)128,
                                     JCSystem.CLEAR_ON_RESET);
```

Why use transient arrays ?

For efficiency, functionality or security:

- "scratchpad" memory
  - RAM is faster & consumes less power than EEPROM
  - EEPROM has limited lifetime
- Automatic clearing of transient array
  - on power-down, and
  - on card reset or applet selection

  can be useful for functionality and/or security

NB there's only a very limited amount of RAM! (in the order of 1 KByte)

Use one or two global transient arrays, allocated once, as scratchpad.

Don't allocate different scratchpad arrays for different purposes.

Programming trick:
transient arrays for session state

```
public class MyApplet {
 boolean keysLoaded, blocked; // persistent state
 private RSAprivateKey priv;

 byte[] protocolState; // transient session state
  ...

 protocolState = JCSystem.makeTransientByteArray((short)1,
                        JCSystem.CLEAR_ON_RESET);
  // protocolState[0] will automatically reset to 0
  // when card powers down & starts up
 ....
```

Remember: Don’t create garbage !

JavaCards have no garbage collector, and very limited memory!

Hence NO CALLS OF

```
new ...

makeTransientByteArray(..)
```

in your code anywhere *except* in the installation phase (e.g. the applet's constructor)

Contents

- Smart Card Operating Systems
- Java Card architecture
- Java vs Java Card
  - APDUs
  - transient and persistent data
  - transactions
  - crypto
- fun with transactions

Transaction example

```
private short     balance;
private short[24] log;
private short     n;
//   log[n-23..n %24] record previous balances
...

  // Update log
  n++;
  log[n % 24] = balance;
  // Update balance
  balance = balance - amount;
```

what if a card tear occurs here ?

Transaction example

```
private short     balance;
private short[24] log;
private short     n;
//   log[n-23..n %24] record previous balances
...
JCSystem.beginTransaction();
  // Update log
  n++;
  log[n % 24] = balance;
  // Update balance
  balance = balance - amount;
JCSystem.commitTransaction();
```

Transactions

- Power supply can be interrupted at any moment, by so-called card tear
- JavaCard VM guarantees atomiticy of bytecode instructions
  (like a normal VM does, except for operations on `doubles`)
- API methods can be used to join several instructions into one atomic action, ie. atomic update of the EEPROM, called a transaction.
  - If power supply stops halfway during a transaction,
    all assignments of that transaction are rolled back.
    This happens the next time the card powers up.

API methods for transactions

- The class `javacard.framework.JCSYstem` provides

```
static void beginTransaction()
static void endTransaction()
static void abortTransaction()
```

- Transactions affects all persistent data in EEPROM (ie. the heap)
  - *except* some very specific EEPROM fields:
    eg. the try-counter fields of PIN-objects

- Transactions do not roll back changes to transient data in RAM (ie. the stack)

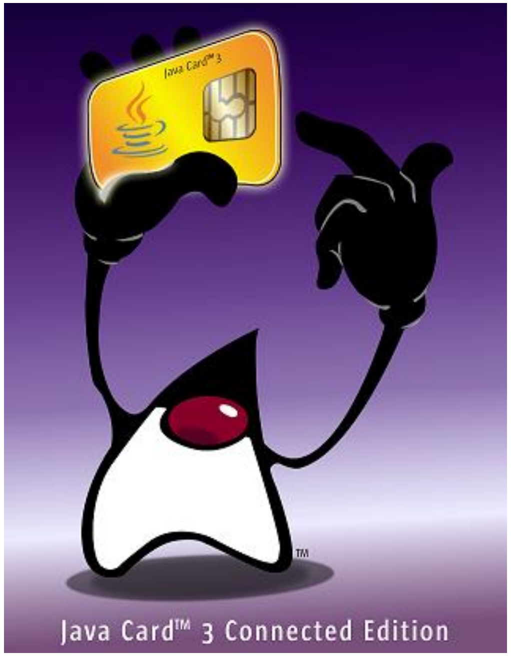
The (weird!) JavaCard 3 Connected Edition

The next-generation smart card OS

- not just arrays, but arbitrary objects can be in RAM
- garbage collection
- multi-threading

  security worries !
- communication with https://

  the smartcard is a web-server!



But who will use it??

intended market: telco

not all card manufacturers are producing JC 3.0 cards

Java(Card) Crypto

Crypto keys in your project code

Do not 'hard code' key material in source code

OR

Document that you have done this in your final report, and then document where

Crypto in Java vs JavaCard

- Beware of confusion between the *different* APIs for crypto in Java and Java Card
- Java Crypto Entension (JCE) provides classes
  - `SecureRandom`
  - `MessageDigest`
  - `Signature and Mac`
  - `SecretKey, PublicKey and PrivateKey`
  - `Cipher` – this is the object that does the crypto work
  - `Certificate`
- Crypto in JavaCard works the same

  except algorithms specified with a short instead of a String

SecureRandom

```
SecureRandom random =
                SecureRandom.getInstance("SHA1PRNG");
random.setSeed(0x3141592653589793L);
...
byte[] output = new byte[8];
random.nextBytes(output);
```

Calling the method `getInstance` may use up memory, so only call this method once, e.g. in the constructor for your applet class, to avoid memory leaks

MessageDigest (ie. a hash)

```
byte[] input1 = ...;
byte[] input2 = ...;
MessageDigest digest = MessageDigest.getInstance("SHA-1");
digest.update(input1);
digest.update(input2);
...
byte[] output = digest.digest(); // returns 20 byte digest
```

Calling the method `getInstance` may use up memory, so only call this method once, e.g. in the constructor of you applet class, to avoid memory leaks

Signed Message Digest (ie. a MAC)

```
byte[] input = ...;
PrivateKey privkey = ...;
Signature signature = Signature.getInstance("SHA1withRSA");
signature.initSign(privkey);
signature.update(input);
...
byte[] output = signature.sign(); // returns 20 byte MAC
```

Such a signed message digest or Message Authentication Code (MAC) ensures integrity of the message.

Calling the method `getInstance` may use up memory, so only call this method once, e.g. in the constructor of your applet class, to avoid memory leaks

Encryption

```
byte[] input = ...;
PublicKey pubkey = ...;
Cipher cipher = Cipher.getInstance("RSA");
cipher.init(Cipher.ENCRYPT_MODE,pubkey);
cipher.update(input);
...
byte[] output = cipher.doFinal();
```

Miscellaneous issues

- **You can generate keys on card and/or in the terminal**
  - **The card can only generate RSA keys in CRT format**
- **To pass keys from terminal to card or vv,
  you need to extract the raw byte arrays from the Key-objects**
  - **Beware that the maximum size of an APDU is limited!**
- **If you create RSA keys in the terminal, beware that the JCE may provide a 1024 bit (= 128 byte) key as a byte array of length 129, because it uses a signed representation, with a leading 0x00.**
  - **remove the leading 0x00 to use such values on the card**
- ***Read the JavaDocs specs of the APIs!***

which eg. say that ALG_RSA_PCKS1 is only suitable for limited length message

Checklist for writing code!!!

- Double-check that there are no calls to

```
new ...
makeTransientByteArray(...)
...getInstance (...)
```

  after personalisation is finished, to prevent memory leaks & running out of memory

- Do not implement your own PIN codes!
  Use the `(Owner)PIN` API class instead

- Are transactions needed to prevent against card tears?

- Can/do you need to store some fields in transient memory to ensure reset upon power-down?

More fun with transactions

Fun with transactions

- **The JavaCard transaction mechanism fundamentally affects the semantics of the language!**

  Presenting the transaction feature in a few apparently simple API calls is a bit misleading...

- **The complexity it introduces can cause major security headaches**

Fun with transactions

```
Class C
{ private short i = 5;

   void m(){

      JCSystem.beginTransaction();
         i++;
      JCSystem.abortTransaction();
      // What should the value of i be?
      //          5, as assignment to i is rolled back


   }
```

Fun with transactions

```
Class C
{ private short i = 5;

   void m(){
      short j = 5;
      JCSystem.beginTransaction();
         i++; j++;
      JCSystem.abortTransaction();
      // What should the value of i be here?
      //        5, as assignment to i is rolled back
      // What should the value of j be here?
      //        6, as RAM-allocated j is not rolled back
   }
```

Fun with transactions

```
Class C
{ private short[] a;

   void m(){
      a = null;
      JCSystem.beginTransaction();
        a = new short[4];
      JCSystem.abortTransaction();
      // What should the value of a be here?
      //          a should be reset to null

   }
```

Fun with transactions

```
Class C
{ private short[] a;

    void m(){
       short[] b;
       JCSystem.beginTransaction();
         a = new short[4]; b = a;
       JCSystem.abortTransaction();
       // What should the value of b be?
       //        null, as creation of a is rolled back

     // Buggy VMs have been known to mess this up...
     // The JCRE specs have since been updated to allow
     // cards to block completely if objects have been
     // allocated in an aborted transaction
    }
```

Fun with transactions

```
Class C
{ private short[] a;

   void m(){
      short[] b;
      JCSystem.beginTransaction();
        a = new short[4]; b = a;
      JCSystem.abortTransaction();
       // Assume buggy VM does not reset b to null
      byte[] c = new byte[2000];
       // short array b and byte array c may be aliased!
       // What will b.length be?
       // What happens if we read/write b[0..999]?
       // What happens if we read/write b[1000..1999]?
}
```

Fun with transactions

```
Class SimpleObject {public short len; }
Class C
{ private short[];
   void m(){
      short[] b;
      JCSystem.beginTransaction();
        a = new short[4];  b = a;
      JCSystem.abortTransaction();
      // Assume buggy VM forgets to reset b to null
      Object x = new SimpleObject();
      x.len = (short)0x7FFF;
     // What will b.length be?
     // It might be 0x7FFF...
}
```

Papers on such attacks

- **Logical Attacks on Secured Containers of the Java Card Platform, Sergei Volokitin and Erik Poll, CARDIS 2016**
- **Malicious Code on Java Card Smartcards: Attacks and Countermeasures, Wojciech Mostowski and Erik Poll, CARDIS 2008**

NB the attacker model is a bit far-fetched:

malicious, possibly ill-typed, applets on a Java Card, that attack other applets or the JavaCard platform via its API, are not a likely attack scenario