Developing secure Java Card applications

Jan Vossaert **Jorn Lapon** **Vincent Naessens**

June 9, 2010

1 Introduction

This tutorial covers the basic steps towards secure smart card application development with the *Java Card platform*. Crucial knowledge of smart cards and the *Java Card platform* for application developers are discussed. Multiple steps during the development of Java Card applications are discussed. Each section also contains one or more practical assignments. All files that are required to execute the assignments are compressed in a `zip` file, namely `javacardtutorial.zip`[1].

1.1 Smart card basics

Java Card technology adapts the *Java platform* for use on devices with limited memory and processing power. A common platform on which the *Java Card technology* is widely used is smart cards. Multiple aspects of smart cards, such as physical characteristics, transmission protocols and their security architecture are defined by the international standard ISO/IEC 7816 [1]. Smart cards are often used in environments with strong security requirements. Examples are electronic identity cards, credit cards, membership cards . . . The card performs certain cryptographic operations such as encryptions and digital signatures. Many smart cards, therefore, provide cryptographic coprocessors that accelerate cryptographic operations. However, the *Java Card platform* is a card-independent high level interface that hides the complexity of the underlying coprocessor and the manufacturer-specific implementation details and technologies.

1.2 Smart card communication

Smart cards communicate with host applications through a request-response protocol in which *application protocol data units* (`APDU`'s) are exchanged. Two types of `APDU`'s exist, namely `C`ommand and `R`esponse `APDU`'s. The former are sent by the host application to the card. The latter are sent by the card, as a response to a `C-APDU`, to the host application.

Mandatory header				**Optional body**		
CLA	INS	P1	P2	Lc	Data field	Le

Table 1: The C-APDU structure.

1. The file can be downloaded from `http://www.msec.be/jan/javacardtutorial.zip`

A `C-APDU` consists of a required header and an optional body, as illustrated in Table 1. The `CLA` byte defines an application-specific class of instructions. According to the ISO7816 standard, byte values between `0xB0` and `CF` can be used. The `INS` byte defines a specific instruction within the class of instructions defined by the `CLA` byte. For valid `CLA` values, the application developer can define his own application specific instructions. The `P1` and `P2` fields can be used to further qualify the instruction and to provide input data respectively. The other fields are optional: the `Lc` field defines the number of data bytes in the data field; the `Data` field can contain up to 255 bytes of data; and the `Le` field defines the maximum number of bytes in the data field of the `R-APDU`.

Optional body	**Mandatory trailer**	
Data field	SW1	SW2

Table 2: The R-APDU structure.

An `R-APDU` consists of an optional body and mandatory trailer. The `Data` field contains the response data, maximum 255 bytes, returned by the applet. The fields `SW1` and `SW2` provide feedback about the execution of the `C-APDU`. Several status words are predefined in the ISO7816 standard. The status word `0x9000` represents successful execution of the command.

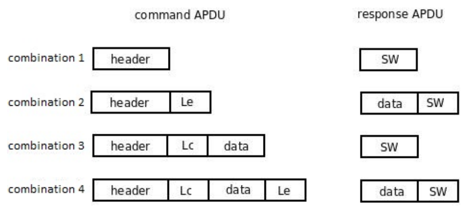


Figure 1: Overview of the possible C/R-APDU combinations.

1.3 Java Card Virtual Machine

The *Java Card virtual machine* (`JCVM`) is split in two components. One component runs at the machine of the developer. It verifies Java classes and converts the Java classes to a `CAP` (*Converted Applet*) file. A `CAP` file keeps all classes in a loadable, executable binary representation. The other component runs at the card. Amongst other things, it interprets the byte code and manages the objects. This separation results in a smaller memory footprint on the card. It, however, introduces additional overhead to application developers.

Due to the memory and processing constraints of smart cards, the `JCVM` only supports a subset of the Java programming language. For example, the types `char`, `double`, `float`, `long` are not supported. Support for `int` is optional. Moreover, the Java core `API` classes and interfaces (`java.io, java.lang` and `java.util`) are not supported except for `Object` and `Throwable`. Further, threads, the security manager and object cloning are not available. For a full overview of the restrictions of the `JCVM`, we refer to the documentation in the `JCDK` [2].

1.4 Java Card Runtime Environment

The *Java Card runtime environment* (`JCRE`) is responsible for card resource management, applet execution, applet security . . . It separates applets from the proprietary technologies of smart card vendors and provides `standard system` and `API` interfaces for applets. The `JCRE` starts executing once the card is inserted in the card reader. It does not resume execution where the previous session ended but starts at the beginning of the main loop. The `JCRE` waits for `APDU` commands from the host. When a command arrives, the `JCRE` either selects an applet – if instructed in the command – or forwards the command to the applet that was already selected at the previous step. The selected applet then takes control and processes the `APDU` command.

The `JCRE` also ensures that each applet is executed in an isolated environment and can, therefore, not influence other applets on the card. This is realized by the – so called – *applet firewall*. It is enforced by the `JCVM` during byte code execution.

The `JCRE` allows for multiple Java Card applets on one smart card. Each applet is uniquely identified by means of an *application identifier* (`AID`). An `AID` is a byte array consisting of around 12 bytes. The structure is defined in the ISO7816 standard.

1.5 TOP IM GX4 and TOP DM GX4

Two types of Java Cards [3, 4] are used in this tutorial. Both are compatible with the *Java Card 2.2.1* and *GlobalPlatform 2.1.1* specifications [5]. The `DM` version provides a dual interface (*contact* and *contactless*) while the `IM` is solely a contact card. They both have an approximate available memory size of 68K and support multiple cryptographic algorithms such as `RSA`, `AES` and `SHA1`.

2 Basic smart card applications

This section introduces the base classes of the *Java Card Framework*, discusses the basic structure of every Java Card applet and provides some guidelines for Java Card development.

2.1 Java Card Framework

The base package of the framework, `javacard.framework`, defines the interfaces, classes and exceptions at the core of the *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 ISO 7816 constants and various Java Card-specific exceptions. The `API` is online available at `http://raud.ut.ee/~tec/static/api/javacard2.1.1/javacard/framework/package-summary.html`. It is also contained in the Java Card Development Kit (see *assignment 1*).

The basic structure of a Java Card applet is given in Listing 1. It extends the `Applet` base class and must implement the `install()` and `process()` methods. The `JCRE` calls the `install()` method when the applet is installed on the card. The `process()` method is called by the `JCRE` upon arrival of an `C-APDU` for the corresponding applet. The `select()` and `deselect()` methods are optional and are respectively called by the `JCRE` upon selection and deselection of the applet. The `select()` method returns `true` if the applet is ready to process incoming

APDU's, or false to decline selection. The deselect() method can, for instance, be used for session termination code.

```
package be.demo;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISOException;

public class Demo extends Applet {

 private Demo() {
 super();
 register();
 }

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

 public void process(APDU arg0) throws ISOException {
  // TODO Auto-generated method stub
 }

 public boolean select(){return true;}

 public void deselect(){}
}
```

Listing 1: Basic structure of a Java Card applet.

2.2 Guidelines for Java Card development

- PIN codes and keys may not be stored in primitive arrays. The Java Card API provides the class OwnerPIN that implements the PIN interface and provides protection against attacks based on program flow prediction. Information about the usage of cryptographic keys can be found in section 6.
- Temporary session data are stored in transient arrays[2]. The API provides methods for creating transient data arrays: `JCSystem.makeTransientByteArray(short length, byte memType)`. This method creates a transient byte array in memory type memType of a certain length. Analogous methods exist for other types of arrays.
- Constants are declared as static final to reduce memory load.
- Instantiate all objects during initialization, typically in the constructor of the applet, to avoid running out of memory at runtime. This also includes the creation of transient objects.

2. Two types of transient memory are available: JCSystem.MEMORY_TYPE_TRANSIENT_RESET and JCSystem.MEMORY_TYPE_TRANSIENT_DESELECT. The former memory type is cleared when the card is removed from the reader while the latter is cleared upon deselection of the applet. Persistent memory holds its values over different sessions.

- Use the Java Card API for transactions if multiple values need to be updated simultaneously: `JCSystem.beginTransaction()`, `JCSystem.commitTransaction()` and `JCSystem.abortTransaction()`. Note, however, that the transaction buffer capacity is finite. The sections that are protected by a transaction should, therefore, only contain the operations that are strictly necessary.

Assignment: *Set up the development environment (Windows).*
Download the `JCDK`[3]. The smart cards in this tutorial support the Java Card 2.2.1 framework. Eclipse[4] is used as `IDE` for the development of Java Card applets. To integrate the functionality of the `JCDK`, the *eclipseJCDE* must be downloaded and installed. This is either done by using the *Install new software* feature[5] of eclipse or by copying the *eclipseJCDE* jar files[6] in the *plugin* directory of eclipse. The plugin can only be used with *Java Cards 2.2.2*. However, after copying (and overwriting) the jar file `org.eclipsejcde.core_0.1.0.jar`[7], the plugin becomes compatible with *Java Cards 2.2.x*. The location of the `JCDK` needs to be specified in order to use the plugin (see figure 2). Finally, the environment variable `JC_HOME` must reference to the `JCDK` main directory (see Figure 3).

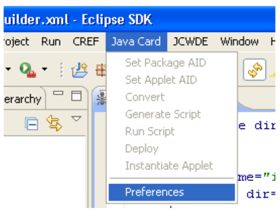
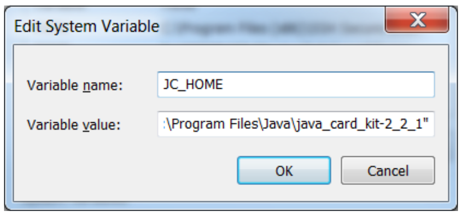


Figure 2: Set location of the JCDK.


Edit System Variable

Variable name: JC_HOME

Variable value: :\Program Files\Java\java_card_kit-2_2_1"

Figure 3: Setting the environment variable `JC_HOME`.

Assignment: *Create Java Card project.*
Java Card projects can now be created through the eclipse wizard (see Figure 4). Call your project *Tutorial*. Use the wizard to create an *IdentityCard* applet in package `be.msec.smartcard`. Select the default values for the applet and package `AID`. Note that the *@Override* annotation, possibly generated by eclipse, is not supported and should, therefore, be removed.

Assignment: *Paste and analyze identity card code.*
An identity card application is used throughout the tutorial for demonstrator purposes. The applet manages an identity file which can be released after the PIN was entered correctly.

3. Different versions of the development kit can be found at: `http://java.sun.com/javacard/downloads/index.jsp`
4. Several Eclipse distributions can be downloaded from `http://www.eclipse.org/`.
5. Choose the *Add site* option and use the following url: `http://sourceforge.net/projects/eclipse-jcde/`.
6. The plugin can be downloaded from `http://eclipse-jcde.sourceforge.net/`.
7. This file is included in `smartcardtutorial.zip`.

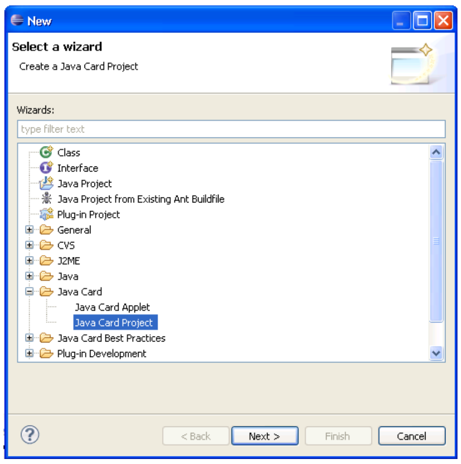


Figure 4: Create Java Card project with the eclipse wizard.

The demonstrator is later extended with signing operations and operations to securely update the identity file. To simplify the design, several attributes – such as keys – are hard-coded in the applet. Replace the code of the previously created applet by the code found in `BasicCardAssignment.txt`. Analyze the new applet.

3 Simulating smart card applications

Before the application is deployed, it can be tested using simulators provided with the `JCDK`.

- The `JCWDE` simulator can run applets in debug mode. However, several cryptographic algorithms are not or only partially available. For instance, several padding schemes are not implemented and RSA keys are limited to 512 bits, while several smart cards support up to 2048 bits. Moreover, it is not possible to store the state of an applet and resume execution later.
- The `CREF` simulator can store the state of applets. Debugging is not possible with this simulator (i.e. no breakpoints can be set).

The `JCWDE` simulator is selected during this tutorial since debugging – which is an important feature – is only supported by this simulator. Using the `CREF` simulator is similar and explained in the Java Card documentation. The `JCWDE` simulator can run in *normal* and *debug* mode, see Figure 5.

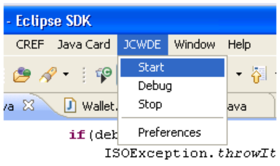
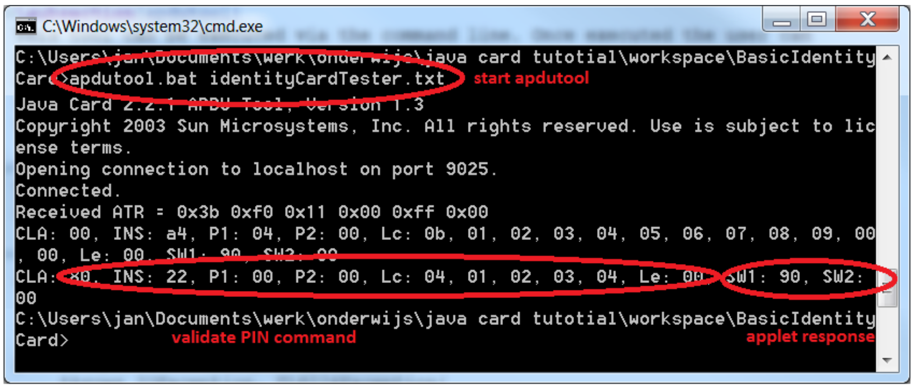


Figure 5: Starting the Java Card simulator

To test the applet, `APDU` commands are sent to the simulated applet. Commands can be sent through the command line – using the `apdutool` – or with a self-written Java program that creates and sends the appropriate `APDU` commands. The latter requires `apduio.jar` in the classpath to enable communication with the simulator.

3.1 apdutool

This tool can be executed via the command line (see Figure 6). The user can enter byte arrays – representing `APDU` commands – and send them to the (simulated) applet. The tool also visualizes the applet's response. Instead of manually entering the `APDU` commands, a script can be passed.



```
C:\Users\jan\Documents\werk\onderwijs\java card tutotial\workspace\BasicIdentity
Card>apdutool.bat identityCardTester.txt
Java Card 2.2.1 APDU Tool, Version 1.3
Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to lic
ense terms.
Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0b, 01, 02, 03, 04, 05, 06, 07, 08, 09, 00
, 00, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 22, P1: 00, P2: 00, Lc: 04, 01, 02, 03, 04, Le: 00, SW1: 90, SW2:
00
C:\Users\jan\Documents\werk\onderwijs\java card tutotial\workspace\BasicIdentity
Card>
```

Figure 6: Using the apdutool to send and receive APDU's.

Assignment: *Test the simulated applet with apdutool.*
Simulate the *IdentityCard* applet and execute the scripts in `apdutoolAssignment.zip` with the *apdutool*. Evaluate the `APDU`'s sent and received from the applet. Add a command to retrieve the identity file from the card.

3.2 apduio.jar

The jar file `apduio.jar` is found in the lib directory of the `JCDK` and enables exchanging `APDU`'s with the simulator in Java programs. An example can be found in Listing 2. In this example a *select applet* command is sent to the applet. The simulator is initialized using the *powerUp* and terminated using the *powerDown* method. Once the simulator is terminated, the state of the applet caused by the prior commands is lost.

```
public void selectApplet(byte[] aid){
  connectToSimulator();
  powerUp();
  //send a select applet APDU
  transmitAPDU((byte)0x00, (byte)0xA4, (byte)0x04, (byte)0x00, aid, 127);
  //... other commands can be send to the simulator here
  powerDown();
}

public Apdu transmitAPDU(byte cla, byte ins, byte p1, byte p2,
    byte[] data, int le) throws IOException, TLP224Exception{
  Apdu apdu = new Apdu();
  apdu.command[0]=cla;
  apdu.command[1]=ins;
  apdu.command[2]=p1;
  apdu.command[3]=p2;
  if(data!=null)apdu.setDataIn(data);
```

```
    apdu.Le=le;
    transmit(apdu);
    return apdu;
}

public void connectToSimulator()
    throws java.io.IOException,java.net.UnknownHostException{
    sock = new Socket("localhost", 9025);
    InputStream is = sock.getInputStream();
    OutputStream os = sock.getOutputStream();
    cad = new CadClient(is,os);
}

public Apdu transmit(Apdu apdu) throws IOException, TLP224Exception{
    cad.exchangeApdu(apdu);
    return apdu;
}

public void powerUp() throws IOException, TLP224Exception{
    cad.powerUp();
}

public void powerDown() throws IOException, TLP224Exception{
    cad.powerDown();
}
```

Listing 2: Creating and exchanging APDU's with applets.

After transmitting the `Apdu` object, it will contain the response bytes of the applet. The status bytes can be retrieved using `apdu.getSW1SW2()`; the method `apdu.getStatus()` returns an `int` representation of the status bytes. If data is included in the response, it can be retrieved as follows:

```
byte[] respData = new byte[apdu.getLe()];
System.arraycopy(apdu.getDataOut(), 0, respData, 0, apdu.getLe());
```

Assignment: *Test simulated applet with apduio.jar.*
Simulate the *IdentityCard* applet and write a Java program in a different project that tests the simulated applet by verifying the `PIN` and visualizing the identity file. The *apduio.jar* file should be added to the project as an *external jar* file. Try setting some break points in the applet code and run the simulator in debug mode. It is also possible to include the standard `JRE` in the applet project. It is then possible to use the functions of the `JRE` such as `System.out.println()`.

Assignment: *Extend the applet with a certificate.*
The sample code can be found in `CertificateTemplate.txt`. The byte array `certificate` contains a hardcoded certificate. Implement the `getCertificate()` method and test the implementation using the simulator. Visualize some fields from the certificate via a Java program.

Some hints are provided. First, since the certificate is larger than 255 bytes, a mechanism that truncates the certificate is required. Two approaches are possible. Either the host application passes to the applet which part of the certificate is requested. This can be done using the `P1` or `P2` fields. The applet can also keep a counter that determines which block of the certificate needs to be sent to the host application. Second, use data blocks of 240 bytes since it then fits in the `APDU` buffer of the simulator. Finally, building a certificate from the `ASN.1 DER` encoding in Java can be done using the following code:

```
CertificateFactory certFac = CertificateFactory.getInstance("X.509");
InputStream is = new ByteArrayInputStream(encodedCert);
X509Certificate cert = (X509Certificate) certFac.generateCertificate(is);
```

4 Deploy smart card applications

Once the applet is written, the *class* files can be compiled (i.e. the *class* files are converted to a `CAP` file). The converter only supports *class* files generated with a Java compiler of a version lower or equal to 4. The converter uses a file that contains several parameters required during the conversion (see Listing 3). It can be executed through the command-line using the command: `converter.bat -config paramFileName`. The `CAP` file can now be passed to the card with the open source program `GPShell`[6]. It uses a script file that contains the necessary references and development keys to deploy applets on smart cards. To automate the whole process, an `Ant` script was written. Some parameters, however, need to be adjusted depending on the project configuration.

```
-out EXP JCA CAP
-classdir .\target
-v
-exportpath "C:\Program Files\Java\java_card_kit-2_2_1\api_export_files"
-applet 0x01:0x02:0x03:0x04:0x05:0x06:0x07:0x08:0x09:0x00:0x00 IdentityCard
be.msec.smartcard 0x01:0x02:0x03:0x04:0x05:0x06:0x07:0x08:0x09:0x00 1.0
```

Listing 3: Example script for generating CAP files.

The `out` parameter defines file types that are generated. The `CAP` file is required to deploy the applet on the card, the others are optional. The `classdir` parameter references to the package where the class files of the applet are stored. The `v` parameter enables verbose feedback about the operations of the converter. The `exportpath` parameter references to the directory where the export files (`.EXP`)[8] of the used framework(s) is/are stored. The `applet` parameter defines the `AID`, the name of the class extending the `Applet` base class, the package name and package `AID`.

```
mode_201
gemXpressoPro
enable_trace
```

8. An export file contains the public API information for an entire package of classes. It defines the access scope and name of a class and the access scope and signatures of the methods and fields of the class. An export file also contains linking information used for resolving interpackage references on the card. An export file does not contain any implementation and can, therefore, be freely ditributed by an applet developer to the potential users of the applet.

```
 4 establish_context
   card_connect
 6 select -AID A000000018434D00
   open_sc -security 3 -keyind 0 -keyver 0 -key 47454d5850524553534f53414d504c45
 8 delete -AID 0102030405060708090000
   delete -AID 01020304050607080900
10 install -file smartcard.cap -priv 04 -sdAID A000000018434D00 -nvCodeLimit 4000
   card_disconnect
12 release_context
```

Listing 4: Example script for deploying applets using GPShell.

Line 2 defines the card type on which the applet will be deployed. Line 7 sets up a secure session with the card; this requires a development key that allows you to place applets on the card. Line 8 and 9 remove the original `AID`s from the card. The previous version needs to be removed if a new version is to be deployed on the card. Note that two `delete` commands are issued. The first one removes the *applet AID* and the second removes the *package AID*. Packages cannot be removed if applets within the package are still on the card. Line 10 installs the applet on the card. This command requires the name of the `CAP` file and the `AID` of the installer applet (`sdAID`). `priv` is an optional attribute. Setting that value `priv` to 4 selects that applet by default. This means that the applet is automatically selected when the smart card is inserted in the card reader. Hence, no *select* command needs to be sent. For further information about the different commands and attributes that can be used, we refer to the documentation that is included with the program. The program also comes with multiple sample scripts for several types of cards.

Assignment: *Deploy IdentityCard applet on a smart card.*
Copy the `Ant` script and additional files found in `deployAssignment.zip` in the *Tutorial* project. The script (`projectBuilder.xml`) requires the other files to deploy the identity applet on your smart card. Some files might require slight modifications (paths to resources), depending on your project configuration, namely `CAPGenerationScript.txt`, `projectBuilder.xml` and `identityCard.txt` (in the `GPShell` directory).

To deploy the applet, run the `Ant` script. When running the script for the first time, toggle the `run configuration` window of the script (see figure 7), select the method(s) of the script that need to be executed (see Figure 8) and set the script to run in the same `JRE` as the project workspace (see Figure 9).

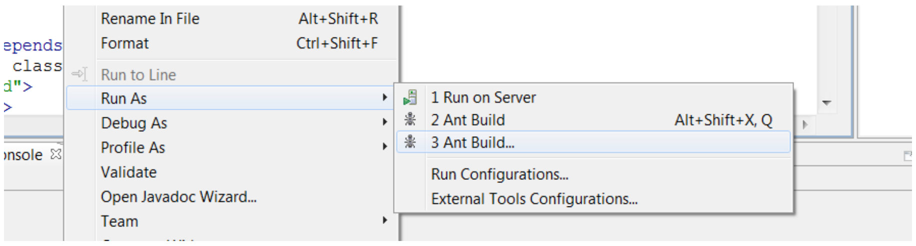
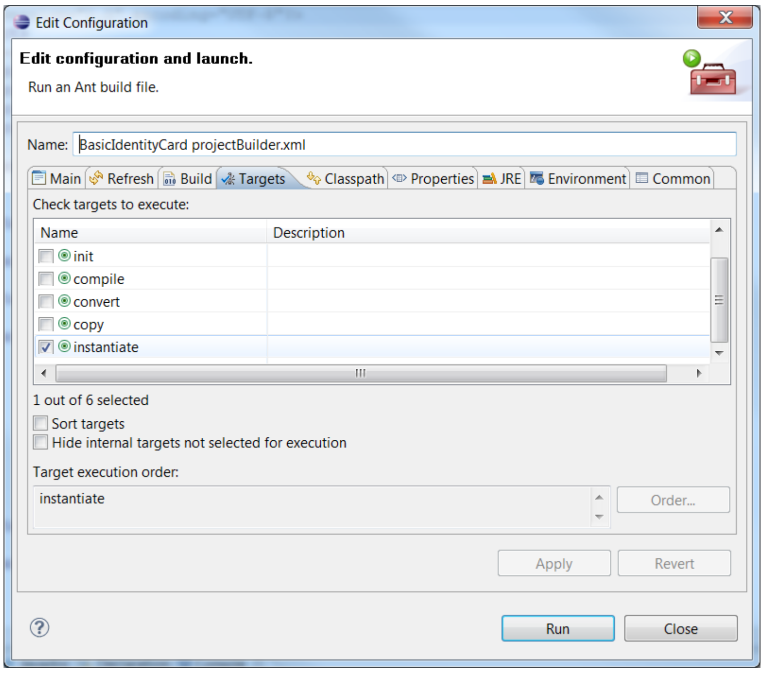


Figure 7: Toggle the 'run configuration' window of the Ant script.


Edit Configuration

Edit configuration and launch.

Run an Ant build file.

Name: BasicIdentityCard projectBuilder.xml

Main | Refresh | Build | Targets | Classpath | Properties | JRE | Environment | Common

Check targets to execute:

Name	Description
☐ init	
☐ compile	
☐ convert	
☐ copy	
☑ instantiate	

1 out of 6 selected

☐ Sort targets

☐ Hide internal targets not selected for execution

Target execution order:

instantiate

Figure 8: Select required method(s) for execution.

5 Writing Java client applications

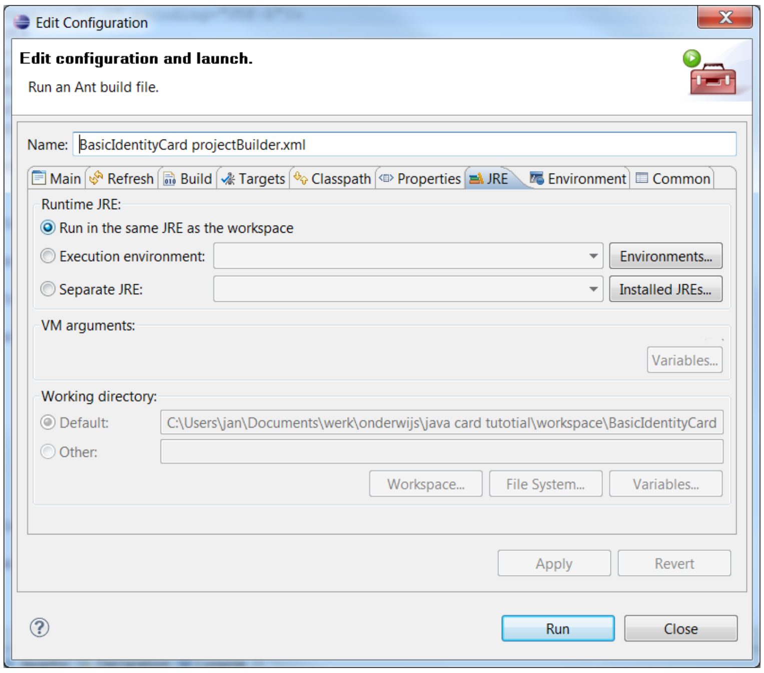
From Java version 1.6, the package `javax.smartcardio` defines an `API` for communicating with smart cards using `ISO/IEC 7816-4 APDU`'s. There are also methods that detect card insertion and card removal. The `CommandAPDU` and `ResponseAPDU` class support the different types of `APDU`'s[9] described in section 1.2.

The listing below shows an example usage of the API for sending and receiving `APDU's`. If the classes from *javax.smartcardio* are not recognized by eclipse, try removing and adding the `JRE`.

```
TerminalFactory factory = TerminalFactory.getDefault();
List<CardTerminal> terminals = factory.terminals().list();
Card card = terminals.get(terminalNumber).connect("*");
CardChannel c = card.getBasicChannel();
byte[] comAPDUBytes = new byte[]{CLA, INS, P1, P2, Lc, data, Le};
CommandAPDU command = new CommandAPDU(comAPDUBytes);
ResponseAPDU resp = c.transmit(command);
byte[] receivedData = resp.getData();
int status = resp.getSW();
```

Assignment: *Write a Java application to test the applet on the smart card.*
Write a Java program that visualizes the identity file stored on the card and some fields from the certificate.

9. The full `API` can be consulted on `http://java.sun.com/javase/6/docs/jre/api/security/smartcardio/spec/javax/smartcardio/package-summary.html`


Edit Configuration

Edit configuration and launch.

Run an Ant build file.

Name: BasicIdentityCard projectBuilder.xml

Main | Refresh | Build | Targets | Classpath | Properties | JRE | Environment | Common

Runtime JRE:

(●) Run in the same JRE as the workspace

( ) Execution environment: [Environments...]

( ) Separate JRE: [Installed JREs...]

VM arguments:

[Variables...]

Working directory:

(●) Default: C:\Users\jan\Documents\werk\onderwijs\java card tutotial\workspace\BasicIdentityCard

( ) Other:

[Workspace...] [File System...] [Variables...]

[Apply] [Revert]

[Run] [Close]

Figure 9: Setting Ant script to execute in the same JRE as the workspace.

6 Cryptographic operations with Java Cards

The `Java Card API` has two packages `javacard.security`[10] and `javacardx.crypto`[11] that contain classes and interfaces for various types of cryptographic algorithms. However, many Java Cards only implement a subset algorithms available in the `API`. More information about the implemented algorithms can be found in the specifications of the manufacturer. The following sections describe how the `Java Card API` can be used to perform cryptographic operations such as encryption, digital signatures and hashing on smart cards.

6.1 Random sequence generation

The `RandomData` class can be used to generate random sequences on the smart card. The security of these sequences depends on the implementation of the smart card manufacturer. Note that `RandomData.ALG_SECURE_RANDOM` is not implemented in the simulator; `RandomData.ALG_PSEUDO_RANDOM`, however, is available.

```
byte[] buffer = new byte[20];
RandomData rand = RandomData.getInstance(RandomData.ALG_SECURE_RANDOM);
rand.generateData(buffer, 0, buffer.length);
```

10. Package contents can be viewed in the `JCDK` or online:
`http://raud.ut.ee/~tec/static/api/javacard2.1.1/javacard/security/package-summary.html`.
11. Package contents can be viewed in the `JCDK` or online:
`http://raud.ut.ee/~tec/static/api/javacard2.1.1/javacardx/crypto/package-summary.html`.

6.2 Key generation

As symmetric keys are just random byte arrays, they can be generated with the `RandomData` class. For asymmetric key pair generation, the class `KeyPair` is used.

```
short keySize = 512;
KeyPair kp = new KeyPair(KeyPair.ALG_RSA, keySize);
kp.genKeyPair();
RSAPrivateKey privKey = kp.getPrivate();
RSAPublicKey pubKey = kp.getPublic();
```

It is also possible to create keys with a predefined modulus and exponent. For this purpose, the `KeyBuilder` class is used. It generates empty key objects which can be initialized by setting the modulus and exponent.

```
short offset = 0;
short keySizeInBytes = 64;
short keySizeInBits = 512;
RSAPrivateKey privKey = (RSAPrivateKey)KeyBuilder.buildKey(
  KeyBuilder.TYPE_RSA_PRIVATE, keySizeInBits, false);
privKey.setExponent(privExponent, offset, keySizeInBytes);
privKey.setModulus(privModulus, offset, keySizeInBytes);

RSAPublicKey pubKey = (RSAPublicKey)KeyBuilder.buildKey(
  KeyBuilder.TYPE_RSA_PUBLIC, keySizeInBits, false);
pubKey.setExponent(pubExponent, offset, (short)3);
pubKey.setModulus(pubModulus, offset, keySizeInBytes);
```

Creating symmetric key objects is similar.

```
short keySizeInBits = 128;
AESKey symKey = (AESKey)KeyBuilder.buildKey(
  KeyBuilder.TYPE_AES_TRANSIENT_DESELECT, keySizeInBits, false);
symKey.setKey(keyData, offset);
```

6.3 Encryption

Multiple algorithms for both symmetric and asymmetric encryption are available. Both are possible with the `Cipher` class. As can be seen in the code listing below, an algorithm identifier is required when creating a `Cipher` object. These algorithm identifiers are defined in the `Cipher` class, examples are: `Cipher.ALG_AES_BLOCK_128_CBC_NOPAD` and `Cipher.ALG_RSA_PKCS1_OAEP`. Note that – if no padding is provided by the cipher (i.e. no `AES` implementation with padding is available) – the input data needs to be block aligned.

```
Cipher cipher = Cipher.getInstance(ALGORITHM, false);
cipher.init(key, Cipher.MODE_DECRYPT);
short encLength = cipher.doFinal(input, inpOffset, length, output, outOffset);
```

Initialization vectors for symmetric encryption in CBC mode can be set with the `cipher.init( key, MODE, iv, offset, length)` method.

6.4 Digital signatures

Although the `Signature` class is mainly used to generate and verify signatures, multiple MAC algorithms are supported.

```
Signature signature = Signature.getInstance(Signature.ALG_RSA_SHA_PKCS1, false);
signature.init(privKey, Signature.MODE_SIGN);
short sigLength = signature.sign(input, offset, length, output, 0);

signature.init(pubKey, Signature.MODE_VERIFY);
boolean val = verify(challenge, chalOffset, length, signature,
  sigOffset, sigLength);
```

6.5 Hashing

The *MessageDigest* class can be used to generate several types of hashes.

```
MessageDigest digest = MessageDigest.getInstance(MessageDigest.ALG_SHA, false);
short hashLength = digest.doFinal(input, inOffset, length, output, outOffset);
```

Assignment: *Extend BasicIdentityCard applet with sign functionality.*
To prove that the card actually contains the `private key` associated with the `certificate`, a sign operation needs to be performed on a challenge that is sent to the card. This operation requires the user to enter his pin. Therefore, implement the *sign* method in the sample applet `SignTemplate.txt`. To test the implementation, simulate and/or deploy the application on a smart card and test it by importing the eclipse project `IdentityCardHost` and execute the `ServiceProvider` class found in package `be.msec.identitycard`. Note that `bcprov-jdk16-143.jar` (included in the zip file) is a required library for the sample application to run. Simulated and deployed applets can be tested using `IdentityCard.SIMULATOR` and `IdentityCard.SMARTCARD` respectively, when requesting an instance from the `IdentityCard` class in the main method. The template contains a sample implementation of the prior assignments and the private key corresponding to the certificate as hard-coded attributes (modulus and exponent of a 512 bit RSA private key are available in byte array representation).
Use the `Signature.ALG_RSA_SHA_PKCS1` algorithm for signatures[12].

Assignment: *Extend BasicIdentityCard applet with update functionality.*
This extension allows to update the identity file remotely (e.g. by the government). A secure session is set up between the server and the card. A new identity file can be sent to the card over the channel. The secure channel ensures data confidentiality, data integrity, and mutual authentication. To realize this, complete the `secureSession`, `getRemainingData` and `updateIdentity` methods in the sample applet of `UpdateTemplate.txt`.

12. `Signature.ALG_RSA_SHA_PKCS1_PSS` is not implemented in the simulator.

The `secureSession` method is called when the government initiates a secure session with the card. The card receives an `APDU` which contains a 20 byte challenge c. The card then composes the following message (where the operator "||" defines a concatenation): `asymEnc(`K_{ses}, PK_{Gov}`)` || `symEnc([sign(c,` SK_{Card}`)` || `cert.length` || `cert` || `padding],` K_{ses}`)`. In the above expression, K_{ses} defines a newly generated 128 bit `AES` session key. Since symmetric keys are random byte arrays, they can be generated using the `RandomData` class. PK_{Gov} defines a 512 bit RSA public key of the government. The byte array representation of the modulus and exponent of this key can be found in the sample applet. SK_{Card} defines a 512 bit RSA private key of the card. The same private key as in the above assignment can be used. The `cert` field defines the certificate of the card. The certificate used in the previous assignment can also be used here. The `cert.length` field defines the byte length of `cert`. Use the `Util` class to convert a `short` to a `byte array`. Since the second part consists of a symmetric encryption with an `AES` key and no cipher implementations for `AES` with padding are available, the encrypted message needs to be block aligned. In `AES`, blocks of 16 bytes are used, which means the byte length of the encrypted message needs to be a multiple of 16. This message exceeds 255 bytes and, therefore, is sent in several steps (see `getRemainingData` instruction). Use the `Cipher.ALG_AES_BLOCK_128_CBC_NOPAD` algorithm for symmetric and the `Cipher.ALG_RSA_PKCS1` algorithm for asymmetric encryption. Although the `Cipher.ALG_RSA_PKCS1_OAEP` is more secure, it is not implemented by the simulator.

As illustrated above, the message consists of two concatenated encryptions. The first one is an asymmetric encryption of the session key with the public key of the government. The second one is a symmetric encryption of (1) a signature on the challenge with the private key of the card, (2) the length of the card certificate, (3) the card certificate and (4) padding. It is important that the message is constructed as indicated above to ensure compatibility with the host application.

The `getRemainingData` method is used to send large amounts of data to the host application. Realize this by maintaining a counter in the applet and implement the `GET_REMAINING_DATA_INS` instruction. This instruction sends consecutive blocks of 240 bytes of the buffered data to the host application.

The `updateIndentity` method receives the new identity file encrypted with the session key. This message is composed as follows: `symEnc([hash(data)` || `data],` K_{ses}`)` where `data` is replaced by: `identityFile.length` || `identityFile` || `padding`. The following operations should be performed on the card upon receiving this message:

- Decrypt the message.
- Check the integrity of the data by comparing the hash (20 byte `SHA1`) with a self-generated `SHA1` hash of the `data`.
- Update the identity file of the card. Use the `Util` class to convert the length field to a `short` and determine the bounds of the identity file.

To test the implementation, simulate and/or deploy the application on a smart card and test it by executing the `Government` class found in package `be.msec.identitycard`. Simulated and deployed applets can be tested using `IdentityCard.SIMULATOR` and `IdentityCard.SMARTCARD` respectively, when requesting an instance from the `IdentityCard` class in the main method.

A sample implementation of this assignment can be found in `finalAssignment.txt`.

7 Requently Occuring Problems

- `I have problems importing the sample application projects in my workspace.` Check that the privileges on the project files allow eclipse to read and write. In some cases the *.project* file is set *hidden*. In that case, make the file visible.
- `I get an error message when converting class files or using the apdutool.` First make sure that the *JC_HOME* environment variable is set to the correct version of the framework. If the error messages still occurs, verify that no program has defined a *CLASSPATH* environment variable (the Belgian governmental eID middleware creates this variable). If so, rename the variable to *CLASSPATHOLD* or redefine this variable (set CLASSPATH=) via the command window prior to execution of the convertor. In the latter solution, the changes effect only the context of the command window in which the command was executed.
- `When converting class files to a CAP file I receive a` ***`unsupported class file format`*** `error.` This error occurs when the class files were compiled with a too recent version of the Java compiler. In eclipse this is easily changed by right clicking on the project, chose *properties* consequently *Java compiler* and then tick the *enable project specific settings* box and change the *compiler compliance level* to 1.4.

References

[1] "Iso/iec 7816." `http://www.iso.org/iso/search.htm?qt=7816&published=on&active_tab=standards`.

[2] "Java card technology." `http://java.sun.com/javacard/`.

[3] "Top im gx4." `http://www.procard.pl/img/site/29/TOP_IM_GX4_Nov07.pdf`.

[4] "Top dm gx4." `http://www.gemalto.com/products/top_javacard/download/TOP_DM_GX4_Product_Information.pdf`.

[5] "Global platform specifications." `http://www.globalplatform.org/specificationscard.asp`.

[6] "Gpshell." `http://sourceforge.net/projects/globalplatform/files/`.

[7] Z. Chen, *Java Card Technology for Smart Cards.* Addison-Wesley, 2000.