smart card and its applications. These are discussed in detail in later sections of this chapter.

Java Card is characterized by the following major benefits:

- *Platform independence.* Java Card applications written in accordance with the specifications are intended to run on any Java Card–compliant smart card. This feature was thought to ensure a high degree of portability of Java Card applications. Unfortunately, individual smart card manufacturers frequently introduce their own packages with a manufacturer-dependent API (especially security-related APIs) or still support different versions of Java Card. This significantly decreases the portability of Java Card applications.
- *Multiple-application support.* More than one application can be run on a Java Card technology smart card. Furthermore, the data of each application is securely protected from any other application run on the same card.
- *Power of Java.* Java Card inherits many benefits of the Java programming language. In the particular case of smart cards, such benefits are object-oriented programming and language-level security. However, some limitations on Java introduced in Java Card (see Section 8.2) frequently lead to a style of programming that is different from conventional Java. Another advantage of Java Card is that its applications can be developed using any development tool or environment for standard Java.

The Java Card architecture is illustrated in Figure 8.1. As can be seen, it looks very similar to traditional Java. The smart card operating system (OS) is layered on top of a smart card microcontroller and is aimed at providing common services like file and data management, communication, and command execution. From the communication point of view, Java Card is fully compliant with ISO/IEC 7816. In particular, Java Card supports communication protocols[1] and commands in accordance with ISO/IEC 7816-3 and ISO/IEC 7816-4, respectively.

The Java Card run-time environment (JCRE) is layered on top of the smart card operating system and consists of the Java Card Virtual Machine (JCVM), the Java Card API, also referred to as the framework, and native

1. T = 0 and T = 1 protocols.

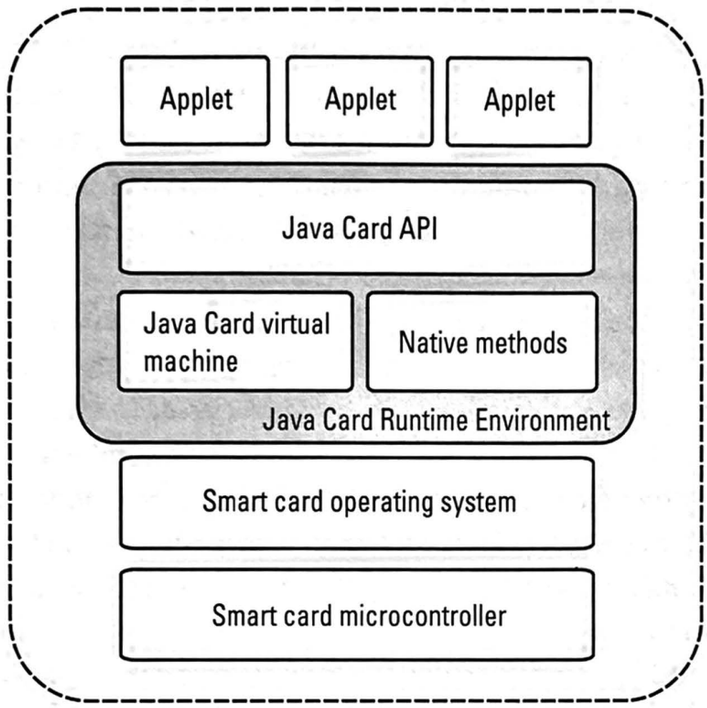

Figure 8.1 Java Card architecture.

methods. Native methods are needed to implement certain special platform-dependent operations like I/O operations or cryptographic operations in a compact and efficient way. That is why the implementation of such operations interacts directly with the smart card operating system and is usually done in languages other than Java (typically, C or Assembler). The Java Card API is formed by a number of packages containing classes dedicated to various purposes (see Chapter 11). In addition to the standard Java Card API, particular JCRE implementations frequently contain some manufacturer-specific extension APIs. On the one hand, they provide some additional functions, but on the other, they decrease the cross-platform portability of Java Card applications.

Java Card applications, called *card applets* or simply *applets,* written in the Java programming language are located on the topmost level of the Java Card architecture. More than one applet can be run on a card. Each applet on a card is uniquely identified by its AID. Chapter 10 of this book addresses security issues involved with the Java Card's multiple-application support.

The main task of the JCVM is to execute an applet bytecode on a card and to provide the Java language support. The core difference between the JCVM and the conventional Java Virtual Machine is that the first one is actually split into two independent parts. One part of JCVM, called the Java Card Converter, is executed off-card, for instance, on a personal computer.

The second part of JCVM is run on-card and is capable of applet code execution, managing classes, and providing interapplet security mechanisms. In contrast to Java, the lifetime of the on-card JCVM is limited only by the lifetime of a smart card. In other words, the on-card JCVM cannot be stopped and then started new again—it always runs on a card and is merely temporarily paused when power is removed from the card.

The Java Card Converter is a software tool that prepares a card applet bytecode (all applet `class` files put into one package) for uploading to a card. This preparation includes verification of classes to be loaded, various checks for Java Card–specific restrictions and violations, allocation and creation of the applet data structures, and resolution of symbolic references to the applet data structures. The result of the conversion is a converted applet (`cap`) file containing a complete image of the applet prepared and optimized for an execution on a card.

Figure 8.2 illustrates the principle described above and shows the main steps of card applet development. A card applet code can be written and compiled using any Java development tool and environment. Debugging and testing is a different case—because of the specifics of Java Card and the use of manufacturer-specific packages, this can be done in most cases only with the help of development tools provided by the smart card manufacturer.

After compilation of all source `java` files related to the applet, the resulting class files are passed to the Java Card Converter, which generates the applet `cap` file as an output. The applet `cap` file then can be uploaded to a card. Java Card specifications do not define exactly how the applet `cap` file is uploaded to a card—this also remains a manufacturer-specific issue.

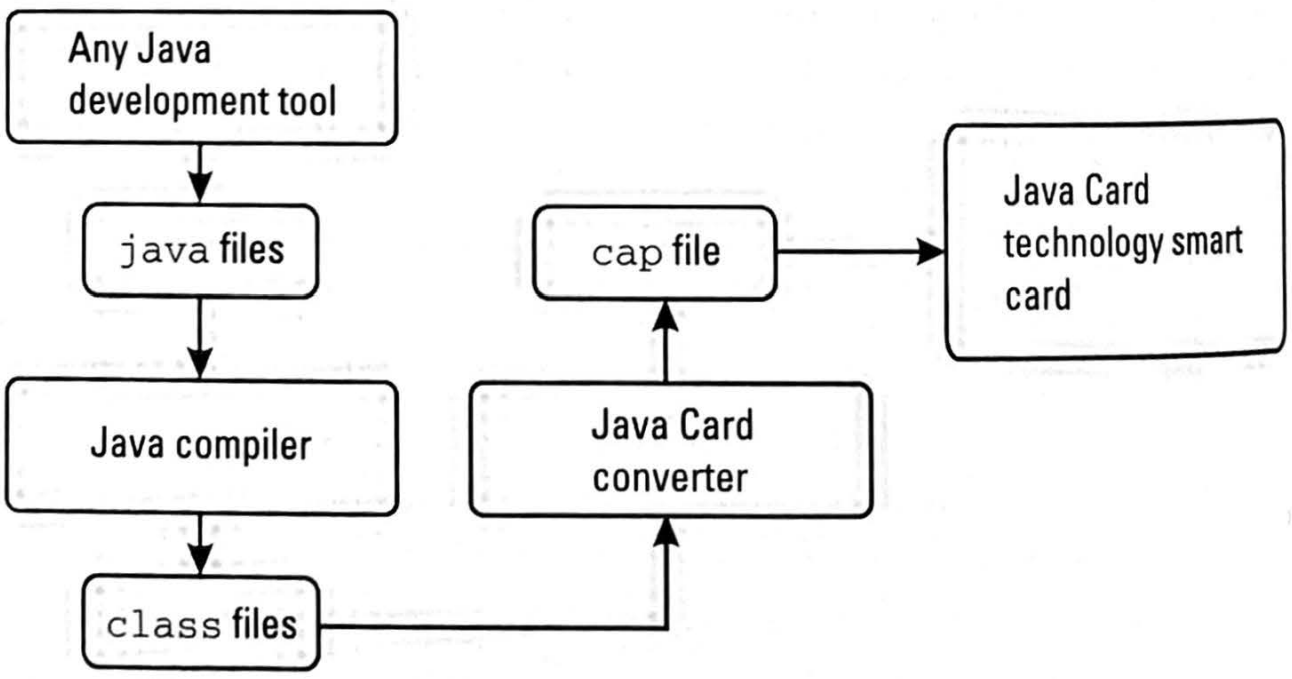

Figure 8.2 Java Card applet preparation.

A few words must be said about how an applet is uploaded to a card. Any Java Card technology smart card contains a special application called the installation program that is capable of loading an applet `cap` file and storing it on the card. Thus, there is no need for JCVM to take care of loading the applet—this is accomplished by the installation program. From an architectural point of view, the installation program can be seen as an ordinary Java Card application layered on top of JCVM and implementing an applet `cap` file upload over certain format APDUs sent to the card by a terminal.

Note that, in order to increase applet uploading security, certain JCRE implementations allow the applet `cap` file to be uploaded in a digitally signed and encrypted manner. In this case, the applet is uploaded successfully only if the applet's digital signature is successfully verified on the card.

Another remarkable feature of Java Card is that it does not provide ISO/IEC 7816-4 file system support on-card. In other words, the Java Card API has no means of working with files in terms of creating, writing, reading, and so forth. All functions related to file representation and handling should be implemented within an applet. Although this looks like a restriction, it gives more flexibility and allows implementation of only those file support features that are really needed by an applet.

Initially, the plan was to provide file system support on Java Card. Even the previous version of Java Card, Java Card 2.0, contained a set of classes dedicated to operations on files. It is said that manufacturers could not come to an agreement on an underlying API and therefore file system support was left out of Java Card 2.1.

A practical object-oriented implementation of a Java Card file system is demonstrated in Part IV of this book.

JCVM, JCRE, and the Java Card API are defined by Sun Microsystems Inc. specifications [1–3], which are available online.[2] As of February 2001, not all existing Java Card implementations were based on Java Card 2.1. For instance, iButton from Dallas Semiconductor and Schlumberger Cyberflex follow the Java Card 2.0 specification.

In May 2000, the Java Card 2.1.1 specification was released [4]. In comparison with Java Card 2.1, Java Card 2.1.1 contains a number of minor improvements and pays more attention to some aspects of Java Card implementation.

---

2. http://java.sun.com/ products/javacard.

8.2 Differences from Java

A smart card is a resource-constrained device. It cannot provide the amounts of memory and high performance that are available on modern computer architectures. That is why it is impossible to implement the standard Java platform in a one-to-one manner on a smart card. The decision was made, therefore, to implement Java Card as a subset of standard Java, omitting some features and adding some restrictions.

First of all, because of the resource constraints and limited CPU performance, Java Card does not support multithreading. Second, Java Card does not support dynamic class loading, for an obvious reason: It is very problematic and almost impossible to ensure loading of additional classes to the card during applet execution. Object cloning is also not supported by Java Card.

All objects once created by an applet will exist as long as the applet exists, that is, until the applet is deleted from the card. This means that all objects[3] created by the applet are persistent, that is, their values are preserved when power is removed from the card. Therefore, Java Card does not need and does not support garbage collection. As a consequence, the method `finalize()` is not supported. This feature also increases applet safety: References to nonexistent objects are avoided because objects cannot be destroyed during an applet's lifetime. On the other hand, implementation of garbage collection could be quite useful in that it could prevent a loss of memory occupied by a dynamic object that leaves the applet's scope. Some Java Card implementations, like iButton from Dallas Semiconductors, support garbage collection.

The following sections discuss in detail certain differences between Java Card and Java.

8.2.1 Primitive Data Types and Arrays

Like Java, Java Card supports such primitive data types as `byte`, `short`, and `boolean`. A byte is an 8-bit signed number with values that can range from –128 to 127. A short is a 16-bit signed number with values that can range from –32,768 to 32,767. A boolean value is represented internally by a byte.

In contrast to Java, Java Card does not support such data types as `float`, `double`, `long`, and `char` at all. Data type `int` is optional; that is, some particular Java Card implementations may support it, some not. A summary of supported and unsupported Java Card primitive data types is given in Table 8.1.

3. Except transient objects that are created in a special manner and whose value is reset upon certain Java Card system events.

Table 8.1
Supported and Unsupported Primitive Data Types in Java Card

Data Type	Width (bits)	Supported?
`byte`	8	Yes
`short`	16	Yes
`boolean`	8	Yes
`int`	32	Optional
`char`	16	No
`float`	32	No
`long`	64	No
`double`	64	No

Java Card supports only one-dimensional arrays, not multidimensional arrays. This limitation is also because of the limited resources available on a Java Card technology smart card. As in Java, elements of an array may be of any supported primitive data type or objects. The following example demonstrates valid declarations of arrays:

```
byte byte_array[] = new byte[3];
byte states[] = {0, 1, 2} ;
PIN app_pins[] = new PIN[3];     // array containing 3
// references to PIN objects
```

The following array declarations are invalid because they declare multidimensional arrays:

```
byte a[][] = new byte[3][3];
boolean flags[][] = new boolean[5][5];
```

As in Java, Java Card arrays are represented by objects. This means that methods of the class `Object` can be applied to them. For instance, an equality of two array references can be checked using the method `equals()` of the `Object` class:

```
if ( states.equals(byte_array) ) {
     ...
}
```

The method returns a boolean value indicating whether the array references are equal or not. More advanced operations on arrays (copying, comparing, etc.) can be performed with the help of static methods of the class `Util`, which is a member of the Java Card framework classes.

8.2.2 Operations and Type Casting

Java Card supports all arithmetic, logical, and bit-wise operations defined in Java. However, typecasting rules used in Java Card are slightly different from rules defined in Java. The main typecasting rule of Java Card states that results of intermediate or unassigned operations must be explicitly cast to a type of a desired value. An intermediate calculation is part of a complex expression involving a number of operations on a number of values. A result of an unassigned operation is not assigned to any variable. An example of an unassigned operation could be an array index calculation.

The reason behind the explicit typecasting rule is that, in Java, results of intermediate or unassigned operations are cast to the type `int` by default. However, Java Card supports the type `int` only optionally, which implies that not all Java Card implementations will have it. Hence, casting either to the types `short` or `byte` must be specified explicitly. The following example demonstrates correct explicit casting of results of intermediate or unassigned operations:

```
byte byte_array[] = new byte[3];
byte b;
short s;
b = byte_array[(byte)(s-1)]; // unassigned operation
b = (byte)( (byte)(s+6)*2 ); // intermediate operation
```

The example below demonstrates erroneous typecasting:

```
b = byte_array[s+1];
b = (byte)( (s+6)*2 );
```

Typecasting errors related to Java Card restrictions are reported by the Java Card Converter.

8.2.3 Exceptions

In principle, Java Card supports all Java mechanisms for exception handling. Card applets may contain `try`, `catch`, and `finally` statements.

Obviously, exceptions related to unsupported features, like multithreading or dynamic class loading, are not supported. Moreover, the constrained resources of a smart card also have an impact, resulting in the following three features of Java Card exception handling:

1. Not all of the Java exception classes are supported.
2. Descriptive string messages in exceptions are not supported. Instead, a reason code of the type `short` is used.
3. Creating instances of exception classes is not recommended. Instead, static JCRE instances of exception classes should be used.

We now discuss each aspect of this list in detail. All Java Card exceptions are subclasses of a superclass `Throwable`. Exception classes are stored in two core packages of the Java Card framework, `java.lang` and `javacard.framework`. Exceptions contained in the first package represent erroneous situations related to Java language programming. Table 8.2 gives a general overview of exceptions contained in the `java.lang` package.

Table 8.2
`java.lang` Package Exceptions

Exception	Description
`ArithmeticException`	Indicates a certain arithmetic run-time error. An example could be the division-by-zero error.
`ArrayIndexOutOfBoundsException`	Indicates that an array index is outside of the array boundaries.
`ArrayStoreException`	Indicates that there was an attempt to store an object of an incorrect type in an array.
`ClassCastException`	Indicates an incorrect attempt to cast an instance of one class to another class.
`NegativeArraySizeException`	Indicates an attempt to create an array with a negative size.
`NullPointerException`	Indicates a null reference access.
`SecurityException`	Indicates a violation of access rights for a certain object.

One important fact must be mentioned: Java Card specifications do not define JCVM behavior for the case in which a certain exception is thrown and is not caught by a card applet. As a first consequence of an uncaught exception, JCVM will halt, that is, card applet execution will be stopped. What will happen then depends on the particular Java Card implementation. For instance, the Sm@rtCafé Java Card technology smart card from Giesecke & Devrient, which is used to implement a sample EMV application later in this book, will respond to a terminal with a status word indicating a general card error.

Exceptions contained in the `javacard.framework` package represent smart card–specific erroneous situations that occur during a card applet execution. Table 8.3 gives their general description.

Java Card does not support the object type `string`. Therefore, Java Card exceptions do not provide descriptive string messages. Instead, additional information about the reason for an exception is reported by a *reason code*. The reason code is a value of the type `short`. A remarkable thing about exception reason codes is that most exception classes, mainly smart card–specific exception classes, contain predefined static constants representing main reason codes typical of the underlying exception.

To conclude the description of Java Card exceptions, a few words must be said about exception usage. First of all, it is strongly recommended not to create a new exception object each time an exception is thrown. Instead, all exception objects needed by an applet should be created during the applet initialization phase, the references to them stored, and the objects reused

Table 8.3
`javacard.framework` Package Exceptions

Exception	Description
`APDUException`	Indicates errors related to APDU handling.
`ISOException`	Is used to issue a response APDU with a given status word.
`PINException`	Indicates errors related to PIN handling.
`SystemException`	Indicates errors occurring on the Java Card at system level.
`TransactionException`	Indicates errors occurring during transaction processing.
`UserException`	Is used to implement user-defined exceptions.

each time an exception must be thrown. In this context, "reused" means that an exception object is created just once but thrown as many times as needed with a desired reason code. The reasoning behind such a practice is obvious: Creating new instances of exception classes will simply waste the limited card memory available.

There is an even more efficient method of exception throwing. JCRE precreates all exceptions defined in the Java Card API. In other words, JCRE creates instances of all Java Card exceptions by default. This means that these precreated exception objects can be used instead of objects created by a card applet, so there is no need to create most of the exception objects at all. All exceptions defined in the `javacard.framework` package (see Table 8.3) have a static method `throwIt()` that throws a JCRE (a precreated) instance of the class.

Let us demonstrate this principle with an example. Assume that an applet must report that the instruction (INS value) given in a command APDU is not supported. This can be achieved with the following statement:

```
ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
```

First of all, execution of this statement will throw a JCRE instance of the `ISOException` exception class with the desired reason code. As a consequence, this exception will force JCRE to issue a response APDU with the ISO 7816-4 status word 6D 00 H defined by the static constant `SW_INS_NOT_SUPPORTED` of the Java Card framework interface `ISO7816`.

8.3 Java Card Applet

The lifetime of a Java Card applet consists of a number of stages. After being compiled and converted to a `cap` file (see Section 8.1) by the Java Card Converter, the applet is loaded to a card by the card installation program. This is the moment when the on-card life of the applet begins. First of all, the applet must be installed and registered within JCRE. If the applet registration is accomplished successfully, the applet becomes available for selection via SELECT APDU, sent to the card, and processed by JCRE. The selected applet is ready to receive incoming command APDUs delivered to it by JCRE, to process them, and to generate response APDUs that are sent out by JCRE.

As pointed out in Section 8.1, the lifetime of a card applet is limited by the lifetime of the Java Card Virtual Machine, that is, by the lifetime of the

card. However, note that certain Java Card implementations may allow clearing of the application area of a card's EEPROM. In this way, all applets existing on the card and all data objects belonging to them are completely deleted from the card.

The Java Card API provides handy mechanisms for card applet implementation. Any card applet is implemented on the basis of an abstract base class `Applet` located in the `javacard.framework` package. The class `Applet` contains all methods necessary for applet installation, selection, and deselection, and APDU processing. Those methods and aspects related to them are discussed in detail in the following section.

8.3.1 Installation and Registration

After an applet has been successfully loaded to the card, it must be installed. The installation procedure is initiated by the INSTALL APDU sent to the card. Java Card specifications do not define the exact format of this APDU; they instead leave it up to the manufacturer. The INSTALL APDU is received and processed by the same card application that loaded the applet `cap` file to the card—the installation program.

On receiving the INSTALL APDU, the card installation program simply invokes a special method of the applet that is to be installed. This method is called `install` and is defined in the abstract class `Applet` extended by any card applet. The installation program also passes to the `install` method applet initialization options received with the INSTALL APDU. The applet `install` method is called only once (obviously, an applet is installed on a card only once).

The core task of the `install` method is to create an instance of the loaded applet class and to register the instance within JCRE. Naturally, the applet constructor is called when the applet instance is created. The constructor may create data objects used by the applet, and it is good programming practice to create all applet objects in the applet constructor.

The applet instance registration is mandatory: If it is not performed, the applet installation fails. The registration is done via invocation of the `register` method of the applet. The `register` method exists in two versions, one with parameters, the other without. The `register` method with parameters is used to specify an AID of the applet instance.

Summarizing everything said above, the main steps of an applet installation procedure (assuming that the applet is already loaded to the card) are as follows:

1. Card installation program receives INSTALL APDU and invokes the `install` method of the applet to be installed.
2. An instance of the applet class is created in the `install` method.
3. The applet instance is registered via invocation of the `register` method.

If the applet is installed successfully, JCRE makes it available for selection.

8.3.2 Selection and Deselection

Any applet installed on a card must be explicitly selected before command APDUs are sent to it. An applet is selected by means of the SELECT APDU with the following defined format:

CLA	INS	P1	P2	L_c	Data
00	A4	04	00	AID length	AID

The data field of the APDU contains an AID of the applet to select. Other fields of the SELECT APDU are fixed and defined in accordance with ISO/IEC 7816-4. If JCRE finds an applet with the given AID, it marks it as selected and forwards it to it all further command APDUs. If no applet with such an AID is found, JCRE reports the fact with the respective status word in the response APDU.

After a card reset, all applets on the card are in a suspended state. In other words, none of the applets is marked as selected. Therefore, if JCRE receives any[4] APDU different from SELECT, it will answer with the response APDU indicating that no applet is selected (status word 69 99 H). Note that some Java Card implementations may allow specification of a default applet. A default applet is marked as selected after a card reset and JCRE will forward to it all received APDUs even if there was no explicit SELECT command. However, Java Card 2.1 specifications address no means for defining a default applet and leave this question up to the manufacturer.

The abstract class `Applet` contains two methods related to applet selection and deselection. The first one is called `select()` and is invoked

4. Except manufacturer-proprietary command APDUs related to card personalization and management, for example, applet load or install APDUs. Command APDUs of this kind are not considered further in this discussion.

by JCRE whenever the applet becomes selected. An applet may perform operations needed for further processing of commands; for example, it may change the values of internal flags. The `select()` method should return a boolean value indicating whether it is ready to accept commands or not. By default, the value `true` is returned.

The applet method `deselect()` is called by JCRE when a currently selected applet becomes deselected, that is, when another applet on the card is selected. Obviously, this method is not called when power is removed from the card.

An interesting feature of SELECT APDU processing is that the APDU is also passed to the applet after its selection by JCRE. This means that the applet also has possibilities of processing this APDU and answering it in a desired manner.

Aspects related to the processing of command APDUs by an applet are addressed in the next section.

8.3.3 APDU Processing

Figure 8.3 demonstrates a general scheme for incoming APDU processing by JCRE. Applet selection mechanisms were presented in the previous section. The abstract class `Applet` extended by any Java Card applet contains the method `process`. This method is invoked by JCRE for each received command APDU. All operations dealing with processing the APDU, performing all necessary application-specific operations in response to the APDU, and preparing the response APDU are done in the applet `process` method.

The `process` method has one single parameter. This parameter is an instance of the `APDU` class, another Java Card framework class located in the `javacard.framework` package. This class provides a handy interface to the communication facilities of a smart card and is designed in a protocol-independent manner. Therefore, an applet developer does not have to deal with specifics of T = 0 or T = 1 protocols (those are the only protocols supported by Java Card 2.1)—all of them are "hidden" inside the `APDU` class and its methods implementation.

A core field of the `APDU` class is a byte array buffer that is used for reading data of the incoming APDU and preparing data of the outgoing (response) APDU. In addition, the class `APDU` provides a number of methods for easy access to the byte buffer.

If no exception is thrown during the `process` method execution, JCRE sends out data in the APDU buffer (if the response was constructed by the applet) with the success status word 90 00 H automatically attached. If

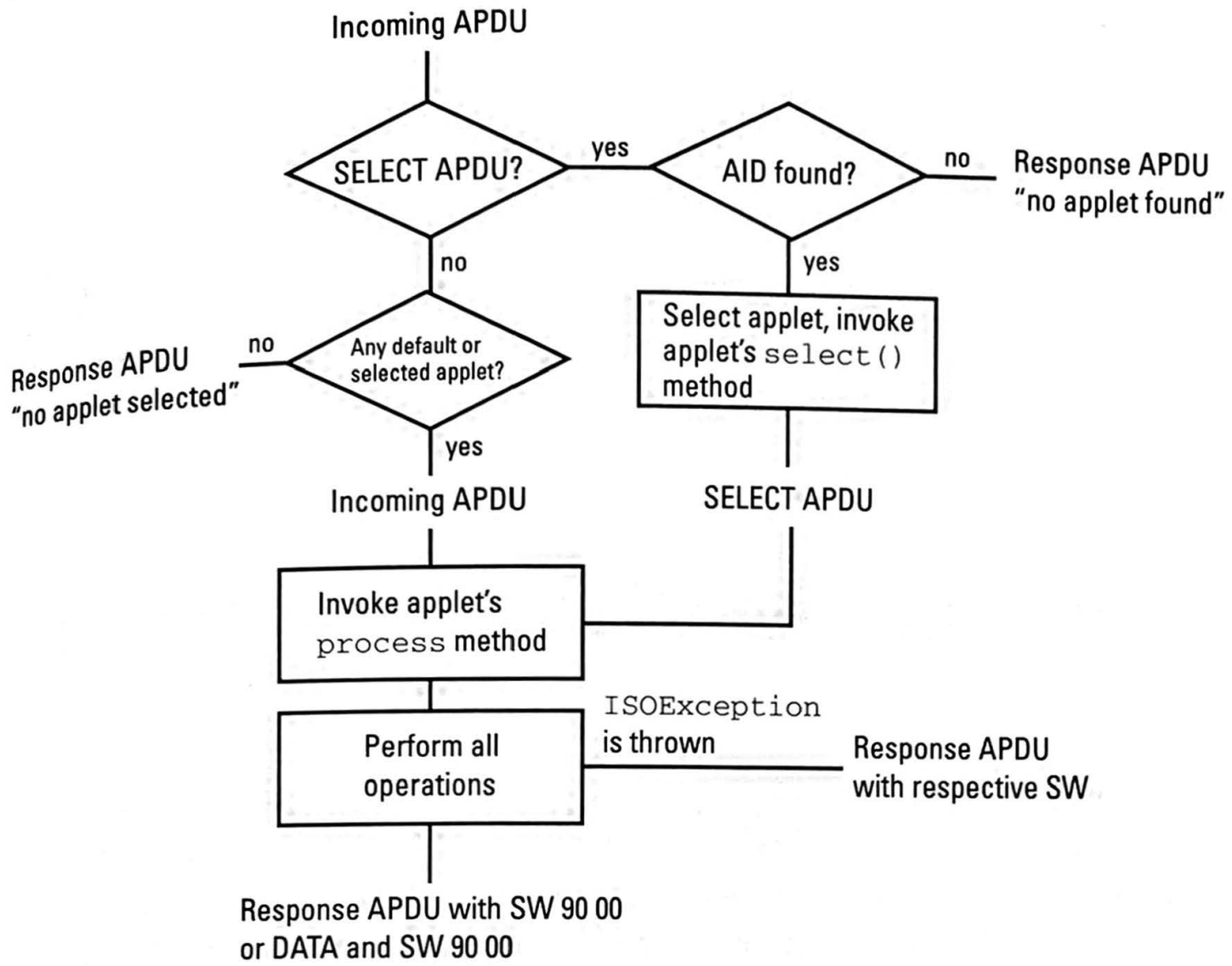

Figure 8.3 Command APDU processing by JCRE and an applet.

the applet throws an `ISOException` (see Section 8.2.3), JCRE catches it and sends out a response APDU with the status word given in the exception reason code. If any other exception is thrown during the `process` method execution, JCRE will send out a response APDU with the status word "No precise diagnosis" 6F 00 H.

The `APDU` class and the `Applet` calls are discussed in Chapter 11.

References

[1] Sun Microsystems Inc., "Java Card 2.1 Virtual Machine Specification," Mar. 1999.

[2] Sun Microsystems Inc., "Java Card 2.1 Runtime Environment (JCRE) Specification," Feb. 1999.

[3] Sun Microsystems Inc., "Java Card 2.1 Application Programming Interface," Feb. 1999.

[4] Sun Microsystems Inc., "Java Card 2.1.1 Specifications. Release Notes," May 2000.

9

Deployment of Java Card Technology

Smart cards can in general be used in mobile phones, personal digital assistants, set-top boxes, and other devices. Java Card technology supports platform independence, makes it possible to implement multiple applications on a single card in a secure way, and allows downloading of applications after a card has been issued. In addition, Java is a programming language in widespread use, which reduces the time-to-market for new smart card applications. All of these properties make Java Card interesting for a range of commercial applications. The following sections give some examples of Java Card technology deployment.

In addition to industry and financial institutions, government agencies have expressed strong interest in Java Card–based products. For example, in 1999, Citibank[1] issued multiple-application smart cards based on Java Card technology to General Services Administration employees. The cards provided a number of functions, including logical access, physical access, property management, e-ticketing, and e-boarding.

9.1 Java Card Forum

The Java Card Forum (JCF)[2] is an interindustry initiative to promote the Java Card API specification as the industry standard. It was founded by

1. http://www.citibank.org.
2. http://www.javacardforum.org.

Schlumberger and Gemplus in 1997 following JavaSoft's (a division by Sun Microsystems) announcement of the Java Card API in 1996. The member list includes chip manufacturers, card manufacturers, companies, and agencies in the financial, telecommunications, health care, transportation, and information technology sectors. Current work is focused on vertical market extensions to the core specification for GSM, banking, and information technology.

9.2 Card Management

As soon as one starts loading and unloading applications to and from smart cards after they have been issued, the problem of managing a card population arises. This is called the card management problem.

The Java Card Management (JCM) Task Force of the JCF was initiated in 1998 with the goal of defining a framework for a card management system (CMS). Specifically, the idea was to define a core CMS on top of which companies could build their own CMS, and to define on-card APIs for Java Card management to be used through an off-card CMS. For example, it is necessary to define the following core features:

- A card repository describing cards with general attributes;
- A card application repository describing the applets with general attributes;
- Card state management functions;
- Life cycle transition management functions;
- Post-issuance applet management.

As of 2001, one package is specified, `org.javacardforum.management` [1]. In addition to this document, several commercial card and application management specifications are available, such as MXI by MAOSCO,[3] Open Platform by Global Platform (see Section 9.4), and the Platform Management Architecture (PMA) by platform7.[4]

3. http:// www.multos.com.
4. http:// www.platform7.com.

9.3 SIM Application Toolkit

Mobile subscriber–relevant data and security algorithms are stored on the SIM (GSM 11.11 [2]).[5] The SIM can be implemented in two forms, either as a smart card or as a plug-in SIM. The SIM card initially played a "passive" role, providing the user with the authentication necessary to access the network and encryption keys to achieve speech confidentiality. SIM Application Toolkit, a part of the GSM standard (GSM 11.14 [3]), extends the card's role such that it becomes the interface between the mobile device and the network. SIM Toolkit supports the development of smart card applications for GSM networks. It is based on the client-server principle, with SMS as the bearer service. In the future, other transport mechanisms such as USSD or GPRS will be used. With SIM Toolkit it is possible to personalize a SIM card, to update existing SIM functions and services, and to install new functions and services by downloading data over the network. This has usually been done by adding or modifying data in the card files and records, not by downloading executable code.

In November 1999, ETSI adopted Java Card technology for inclusion in SIM Toolkit [4]. In the same year, ETSI issued a standard (GSM 03.19 [5]) describing the following extensions to the Java Card 2.1 API:

- The `sim.access` package provides the means for the applets to access the GSM data and file system of the GSM application defined in the GSM 11.11 specification.
- The `sim.toolkit` package provides the means for the toolkit applets to register the events of the toolkit framework, to handle TLV (tag-length-value) information, and to send proactive commands according to the GSM 11.14 specification.

The resulting cards provide GSM operators with the ability to deploy a wide range of value-added services, such as secure remote banking, stock trading, and unique dial-back roaming services. There are already Java Card 2.0–based SIM cards on the market, such as Giesecke & Devrient's StarSIM, Gemplus's GemXplore98, or Schlumberger's SIMera. Card applets can usually be transported to the card by SMS, either from a content provider or at a point-of-sale terminal. Cards have a Java Virtual Machine that supports the

5. GSM standards are issued by the European Telecommunications Standards Institute (ETSI); see http://www.etsi.org/.

sandbox security model, strong bytecode verification, and firewalls between card applets.

9.4 Visa Open Platform

An interesting development in the smart card and e-commerce area is the Visa Open Platform [6] supported by various financial institutions, service providers, mobile network operators, and hardware manufacturers. The goals are to develop standardized solutions for secure mobile electronic commerce and also an Open Platform chip that will allow financial institutions to dynamically download Visa payment applications to a mobile phone on the basis of Java Card technology. The technology is chosen in such a way that it ensures these goals will be reached:

- Interoperability of cards, terminals, operating systems, software products, and bank office support systems from different vendors;
- Secure support of multiple applications coexisting on the card (Java, Java Card, Windows for Smart Cards) in such a way that each application provider is assigned a separate security domain;
- Strongest commercially feasible security, which will be evaluated using the Common Criteria (see Chapter 3);
- Support of existing standards such as EMV (see Section 7.1) and ISO 7816 (see Section 1.5) so that the card can be used in the existing ISO/EMV-compliant terminals.

An Open Platform card could serve as a corporate credit card, stored-value purse for small purchases, security token for Internet commerce, or an electronic ticket carrier. Two specifications are relevant to smart cards:

1. The Open Platform Card Specification specifies the off-card communication with the terminal and the on-card communication with the applications. In other words, it defines the Open Platform API and how to use it to develop card applications.
2. The Visa Open Platform Card Specification defines the enhancements to the Open Platform that are needed to implement Visa-specific applications (e.g., cryptography support).

In the run-time environment, two different stacks are possible. One stack includes the Java Card Virtual Machine with the Java Card API layered over a proprietary card vendor operating system. Another stack includes Windows for Smart Cards (WfSC) with the corresponding Virtual Machine and API, layered over the WfSC operating system. In addition, the stack includes the Open Platform API, which extends the standard card API to allow additional security control (e.g., secure channel establishment, key verification before loading it on the card, card lock if a security threat is detected). The Open Platform environment also places some additional constraints on applications (e.g., secure card auditing, application loading after a card has been issued). For example, if an application is loaded after the card is issued, a secure channel is established between the card and the platform from which the application is loaded. In this way the card can authenticate the application provider, and the integrity of the application is guaranteed. The Open Platform defines its own card management (see also Section 9.1).

References

[1] Java Card Management Task Force, "Java Card Management Specification," Version 1.0b, Oct. 2000; available at http://www.javacardforum.org/Documents/documents.html.

[2] European Telecommunications Standards Institute, "Digital Cellular Telecommunications System (Phase 2+); Specification of the Subscriber Identity Module—Mobile Equipment (SIM-ME) Interface (GSM 11.11, Version 8.3.0 Release 1999)," 2000.

[3] European Telecommunications Standards Institute, "Digital Cellular Telecommunications System (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module—Mobile Equipment (SIM-ME) Interface (GSM 11.14, Version 8.3.0 Release 1999)," Aug. 2000.

[4] Hassler, V., *Security Fundamentals for E-Commerce,* Norwood, MA: Artech House, 2001.

[5] European Telecommunications Standards Institute, "Digital Cellular Telecommunications System (Phase 2+); Subscriber Identity Module Application Programming Interface (SIM API); SIM API for Java Card (TM); Stage 2 (GSM 03.19, Version 7.1.0 Release 1998)," May 2000.

[6] Visa International, "Visa Open Platform: Overview," 2000; available at http://www.visa.com/nt/suppliers/open/overview.html.

10

Java Card Security

Smart card security issues can be divided into four areas: (1) card body security, (2) hardware (i.e., chip) security, (3) operating system security, and (4) card application security. In addition to these general issues, which are addressed in Part I of this book, Java Card security encompasses the following areas: Java Card language subset security, card applet security mechanisms, and Java Card crypto APIs for writing secure programs. This chapter gives a brief overview of these aspects of Java Card.[1] For more details, please refer to [1].

10.1 Java Card Language Subset Security

As of mid-2001, Java was probably the most popular programming language [2, 3]. Its development started in 1991 at Sun Microsystems when James Gosling developed the Oak programming language. Oak was designed for consumer electronics software that could be downloaded (i.e., upgraded) over a network. The programs written in Oak were supposed to be very compact and highly reliable. Because portability (i.e., platform independence) was one of the major design goals, the source code was compiled into an interpreted bytecode to run on a virtual machine. In other words, the Oak bytecode contained a set of instructions not typical of any particular microprocessor, but for a specially designed "virtual microprocessor" (virtual machine).

1. http://java.sun.com/ products/javacard.

Java is a general-purpose object-oriented programming language similar to C++. It began from a subset of C++ in which all features considered error prone or unsafe were eliminated [4]. Some of Java's object-oriented properties are dynamic binding, garbage collection, and inheritance. Java programs are compiled into a processor-independent bytecode, which is loaded into a computer's memory by the Java Class Loader to be run on a Java Virtual Machine (JVM). JVM can run programs directly on an operating system or be embedded inside a Web browser. It can execute the Java bytecode directly by means of an interpreter, or use a "just-in-time" (JIT) compiler to convert the bytecode into the native machine code of the particular computer. JVM enforces Java safety, privacy, and isolation rules. These make it possible to protect against unauthorized access and to isolate one application from another within the same address space, so that it is not necessary to enforce address space separation between applications [5].

As a subset of the Java programming language and virtual machine specifications, the Java Card platform inherits the main Java security features such as Java safety and Java type safety, which are briefly described in the following two sections.

10.1.1 Java Safety

The term *safety* denotes the absence of undesirable behavior that can cause system hazards. Java is a safe programming language: Many of the confusing or poorly understood features of C++ cannot be found in it. For example, Java manages memory by reference and does not allow pointer arithmetic. Another feature that makes Java simpler and thus safer is that it does not allow multiple class inheritance. On the other hand, Java allows multiple interface inheritance. However, an interface, in contrast to classes, may not be used to define an abstract data type, since it may contain only constants and method declarations, and no implementations. Java also provides the `final` modifier, which disables subclassing when applied to class definitions and disables overriding when applied to method definitions.

In addition, some new mechanisms that can be programmed in C++ only by very experienced programmers are a part of the language in Java. For example, a useful mechanism is exception handling, which can be employed by a programmer to specify how a program should manage an error condition. If a Java program tries to open a file that it has no privilege to read, an exception will be thrown, but the program will not abort. Some of the security-related problems in other languages resulted from programming faults, but the fact that Java is safe cannot protect executing hosts against

intentionally malicious programs or smart cards against malicious card applets [5].

10.1.2 Java Type Safety

Java is a strongly typed language. This effectively means that an object must always be accessed in the same way, so that illegal type casting is impossible. By using a cast expression it is possible to instruct a compiler to treat, for example, an integer as a pointer, or a pointer to one type as a pointer to another type. In Java, it cannot happen that one part of the program sees an object as having one type, and another part of the program sees that object as having another type.

Java employs both static and dynamic type checking. Pure dynamic type checking is the safest way to perform type checking. It can be done by checking an object's tag before every operation on it to make sure that the object's class allows such an operation. Unfortunately, dynamic type checking makes programs run slowly. Therefore, Java also employs static type checking, which is much more complicated but can be performed before program execution (i.e., only once). If Java can determine that a particular tag-checking operation will always succeed, then there is no reason to check it dynamically. Static (or load-time) type checking is performed by the bytecode verifier and ensures that the program does not forge pointers, violate access restrictions (i.e., public, protected, private), violate the type of any object, try a forbidden type conversion (illegal casting), or contain stack overflows.

Static checking is performed by the off-card JCVM. Dynamic (or run-time) type checking is performed by the on-card JCVM (i.e., JCRE) and ensures that there are no array boundary overflows or type incompatibilities. Type safety has direct implications on Java security [6].

As pointed out in [7], it would be rather difficult to prove type soundness for Java. Type soundness is based on specifying all possible behaviors that a well-typed program can exhibit, basically by enumerating all errors that may cause the program to abort according to the programming language semantics. In Java many possible reasons exist for run-time errors (e.g., invalid class format), and Java programs may, under some circumstances, terminate in unexpected ways (i.e., cause a segmentation violation).

10.1.3 Transient Objects

Temporary data can be stored in transient objects in RAM. Their contents are set to a default value (e.g., `NULL` or `false`) at the end of their lifetime. If

the lifetime is defined as `CLEAR_ON_RESET`, a transient object's contents are set to a default value when the card is reset. If the lifetime is defined as `CLEAR_ON_DESELECT`, a transient object's contents are set to a default value when the applet is deselected. This feature is very important for security parameters such as the PIN, session keys, or private keys. If such parameters were stored as persistent objects and were not explicitly cleared before the applet was deselected, the applet to be selected next would be able to read their values.

10.1.4 Atomicity of Transactions

An e-payment transaction must be atomic, meaning that it is either fully performed or not at all. In other words, it must not remain in an undefined state. For example, consider the situation in which a card is pulled out of a card reader in the middle of a payment transaction just before the balance on the card is updated but after a valid payment message has been sent to the payee. Without atomicity, this would imply that the payee received the money and would deliver the goods, but the payer's card balance was not reduced. With atomicity, this transaction would simply be aborted.

Java Card supports a transaction model in the following three ways [1]:

1. A single update to a field of a persistent object or a class is always atomic. If an error occurs during update, the content is restored to its previous value.
2. Block updates of multiple data elements in an array are atomic if the `arrayCopy` method is used.
3. An update of several different fields in different persistent objects performed by an applet can be atomic so that either all updates take place or all fields are restored to their previous values.

10.2 Card Applet Security Mechanisms

There are basically two types of applets [1]:

1. Preissuance applets' classes are burned (or "masked") into ROM at the same time as the JCRE during the manufacturing phase of the card life cycle. They are also called ROM applets. Preissuance applet instances are instantiated in EEPROM by the JCRE. Because they are provided by the card issuers (i.e., by a trusted

source) they may declare native methods. Native methods are written in another programming language and are not subject to Java security checks.

2. Postissuance applets' classes can be downloaded (e.g., into EEPROM) onto the card after the manufacturing phase. For security reasons they are not allowed to declare native methods because their content and behavior cannot be controlled by the JCRE.

An applet can register itself with the JCRE by its AID (applet identifier). Card applet authentication is usually based on the AID, but for improved security it is recommended that a cryptographic mechanism be used in addition. For example, all postissuance applets may need to be signed by the origin so that the digital signature of the `cap` file can be verified before downloading.

The following sections explain two important card applet security mechanisms: an applet firewall enforced by the JCRE and secure object sharing among applets.

10.2.1 Card Applet Firewall

One of the main advantages of Java Card is that it can host multiple applications, that is, multiple applets can reside on one card. This feature, however, raises security issues of code and data sharing, or in other words, the issues of controlling access to code and data on the card. Applets should not be able to access each other's data. For example, no cardholder would be happy if the tax collecting application on his Java Card could read data from his personal bookkeeping application. Therefore, the Java Card has a mechanism called an *applet firewall,* which means that applets cannot access each other's data unless they explicitly allow it through the `Shareable` interface. PIN-based cardholder authentication is also supported.

The applet firewall is also a Java Card run-time security check, in addition to Java Card language subset type safety checks (see Section 10.1.2). The "normal" Java programming language allows access to public methods even across packages. The Java Card introduces the concept of a *context,* which represents a separate object space shared by all applets belonging to the same package. Because of the firewall mechanism enforced by the JCRE, an applet may not access objects from a different context. The JCRE has access to all applets and objects created by applets (i.e., all contexts), and all applets have access to global arrays owned by the JCRE, such as the APDU buffer.

Applets gain access to JCRE services through JCRE entry point objects. This means that the public methods of such objects may be invoked from any context. References to temporary JCRE entry point objects (for example, APDU objects or JCRE-owned exception objects) cannot be stored by the invoking applet. References to permanent JCRE entry point objects may be stored and reused. Examples are AID instances created by the JCRE to encapsulate an applet's AID when the applet instance is created [1]. Global arrays are a special type of JCRE entry point object. By using them, applets from different contexts may share only primitive data. The shareable interface mechanism explained in the next section makes object sharing possible among applets from different contexts.

10.2.2 Secure Object Sharing

The first Java Cards based their applet data-sharing policy on access control lists. An access control list defined for each identity which item it could access and with which particular access permissions (e.g., read, write). The items were files, and the identities were defined by means of key files and PINs. That approach did not, however, allow object methods to be shared between different applets, but only data. In other words, it was not possible for an applet to invoke another applet's method [8]. The means of sharing objects between applets was introduced by the Java Card 2.1 specification.

Basically, the Java Card 2.1 object-sharing mechanism also uses access control lists, but this time the identities are established through the unique applet identifiers (AIDs). A card applet with a specific AID may obtain an interface belonging to another applet and thus invoke its methods.

The following explanation of the Java Card object-sharing mechanism is based on [8]. If an applet instance (server applet) wishes to share some methods with applets from different contexts, with the Java Card 2.1 API it does the following:

- The server applet defines a shareable interface PI extending the interface `javacard.framework.Shareable`.
- The server applet defines a class PC implementing the shareable interface.
- The server applet creates an instance PO of class PC.
- The server applet registers with the JCRE by submitting its AID.

Object PO is referred to as the *shareable interface object* (SIO). This mechanism was introduced by the Java Card 2.1 specification. When an applet instance (client applet) wishes to access object PO from the server applet, it performs the following steps:

1. The client applet creates an object reference CO of type PI.
2. The client applet calls a system method `getAppletShareableInterfaceObject(ServerAID, byte)` with the AID of the server applet and with an optional byte carrying the identifier of the selected interface (if more than one is provided by the particular server applet). The JCRE forwards the request to the server applet with the first argument replaced by the client applet's AID.
3. When the server applet receives the request, it makes its access control decision based on the client applet's AID; if the client is permitted to share object PO, the server returns a reference to PO (of type SIO), otherwise it returns a `null` to the JCRE.
4. The JCRE forwards the object reference to the client applet; the client casts the object reference to type PI and stores it in CO.

Now when the client applet invokes a method on CO, a context switch is triggered in the JCRE. This means that because of the applet firewall the client can see only the object SO, and the server can see only the arguments passed on the stack (as well as the APDU buffer).

This object sharing model does, however, have some serious security problems:

- *AID spoofing.* Access control decisions made by the server applet are exclusively based on AIDs. If a malicious and fake applet has the AID falsely set to be the same as a client applet known to the server and it (instead of the genuine client applet) is loaded onto the card, it may gain access to the shared interface. The solution to this problem is to allow loading of only applets signed by a trusted source.

- *Inflexible access control.* Because access control is based on AIDs, a server applet must know in advance (i.e., before being loaded onto the card) the AIDs of all applets with which it will share objects. If an applet to share an interface with is written after the server applet has been loaded, there is no flexible way to add the new AID to the server applet's access control list.

- *Illegal reference casting.* Suppose a server applet shares interface PI1 with an applet specified by AID1, and interface PI2 with an applet specified by AID2. Client applet AID1 could, after legitimately obtaining interface PI1, cast interface PI1 into interface PI2 and thus gain access to methods not intended to be shared with it by the server applet. In [8] two work-arounds are proposed: (1) to use a separate delegate object for each shared interface that redirects calls to the intended object, or (2) to check the AID of a client applet each time it tries to access a server applet's method.

- *Inability to pass object parameters.* The only way to pass object parameters between the server and the client is to use the APDU buffer (i.e., global array; see Section 10.2.1). This approach is sometimes very inconvenient because the data to be passed must first be converted into a representation suitable for this type of exchange. Unfortunately, allowing applets to access objects including their data and not only interfaces could potentially open up new security holes.

10.3 Java Card Crypto APIs

Java Card cryptography APIs are based on the Java Cryptography Architecture (JCA),[2] which represents a framework for accessing and developing cryptographic functionality for the Java platform. Because of U.S. export regulations on cryptography it was necessary to provide algorithm extensibility and independence so that different cryptographic algorithms could be implemented by the JCRE providers. In addition, implementation interoperability ensures that applets can access cryptography services on the card without knowing the actual name of the implementation class. This is achieved by factory methods and naming conventions for specifying algorithms and their parameters. For example, SHA can be specified as `MessageDigest.ALG_SHA`; an instance of a class implementing SHA can be obtained by calling the factory method `getInstance(algorithm, externalAccess)` of the `MessageDigest` class. The `algorithm` parameter is set to SHA, and the `externalAccess` parameter can be set to true or false. If it is set to true, the `MessageDigest` instance may be shared among multiple applet instances and it is accessible via a `Shareable`

2. http://java.sun.com/products/jdk/1.3/docs/guide/security/CryptoSpec.html.

interface when the owner of the instance is not the currently selected applet (see also Section 10.2).

The two crypto API packages are `javacard.security` and `javacardx.crypto`. The `javacard.security` package contains interfaces for implementing the following:

- Symmetric and asymmetric keys (`Key`, `SecretKey`, `DESKey`, `PrivateKey`, `PublicKey`, `RSAPrivateKey`, `RSAPrivateCrtKey`, `RSAPublicKey`, `DSAKey`, `DSAPrivateKey`, `DSAPublicKey`, `KeyBuilder`, `KeyPair`);
- Authentication (`MessageDigest`, `Signature`);
- Random data generation (`RandomData`);
- Crypto exceptions (`CryptoException`).

The classes in the `javacardx.crypto` package (`Cipher`,`KeyEncryption`) are subject to U.S. export control (strong encryption).

10.4 PIN Verification

To prevent unauthorized use of a smart card, the user is usually required to enter a PIN, an alphanumerical string having six to eight characters at most (otherwise it would be too difficult for the user to remember it). The card owner types his PIN on a PC keyboard or on a keypad on the card reader (i.e., CAD). The keypad is more secure because it is not possible on the PC to intercept the PIN from the keyboard strokes. The card locks after a certain number (e.g., three) of unsuccessful attempts to enter the right PIN. PIN initialization is performed at applet creation and installation [9]. This means that it is possible to define a different PIN for each application (i.e., applet) on the card.

The PIN is represented by a public `PIN` interface in the `javacard.framework` package. An implementation maintains the following values:

- PIN value;
- Maximum number of unsuccessful attempts allowed;
- Maximum PIN length;
- Remaining number of unsuccessful attempts allowed;
- Validated flag (true if a valid PIN has been presented).

References

[1] Chen, Z., *Java Card Technology for Smart Cards,* Reading, MA: Addison-Wesley, 2000.

[2] Gosling, J., B. Joy, and G. Steele, *The Java Language Specification,* Reading, MA: Addison-Wesley, 1996.

[3] Lindholm, T., and F. Yellin, *The Java Virtual Machine Specification,* Reading, MA: Addison-Wesley, 1997.

[4] MageLang Institute, "Fundamentals of Java Security," Jan. 2000; available at http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/index.html.

[5] Hassler, V., *Security Fundamentals for E-Commerce,* Norwood, MA: Artech House, 2001.

[6] McGraw, G., and E. Felten, "Java Security and Type Safety," *Byte,* Vol. 22, No. 1, 1997, pp. 63–64.

[7] Volpano, D., and G. Smith, "Language Issues in Mobile Program Security," *Mobile Agents and Security,* G. Vigna (ed.), LNCS 1419, Berlin: Springer Verlag, 1998, pp. 25–43.

[8] Montgomery, M., and K. Ksheerabdhi, "Secure Object Sharing in Java Card," *Proc. USENIX Workshop on Smartcard Technology,* Chicago, IL, May 10–11, 1999; available at http://www.usenix.org/publicaitons/library/proceedings/smartcard99/montgomery.html.

[9] Chen, Z., "How to Write a Java Card Applet: A Developer's Guide," *JavaWorld,* July 1999; available at http://www.javaworld.com/jw-07-1999/jw-07-javacard.html.

11

Application Development

This chapter gives an introduction to Java Card application development, beginning with an overview of core classes and methods of the Java Card API. The API is explained on a general level. Readers interested in details of particular classes or methods should refer to the Java Card API reference manuals. The section concludes by presenting Java Card implementations currently available on the market.

11.1 Java Card API

The Java Card API , also referred to as the *framework*, consists of four core packages. Two packages, called `java.lang` and `javacard.framework`, were mentioned in previous sections of the book. The `java.lang` package contains all classes related to support of the Java programming language subset. The `javacard.framework` package contains classes related to Java Card applet functionality.

The other two packages, `javacard.security` and `javacardx.crypto`, play a particularly important role for several reasons. First of all, they contain classes related to the security functionality of an applet. This security functionality mainly covers the cryptographic API and its support. Unfortunately, implementation of the security-related Java Card classes is still not standardized among smart card manufacturers. Usually, smart card manufactures provide security-related classes within their own proprietary packages, which are shipped in a form of an extension API. Therefore, their

description is omitted in the sections that follow (see also Section 10.3). However, a closer look at manufacturer-specific issues of the Java Card API will be taken in Section 11.2, where existing Java Card implementations are presented.

The following sections provide an overview of the core classes of the `javacard.framework` package.

11.1.1 `JCSystem` Class

The `JCSystem` class contains a number of methods for applet execution control, object management, and atomic transaction support. All methods of the class are static. A rather large group of the class methods is used for performing atomic transactions (see Section 10.1.4). The methods `beginTransaction()`, `abortTransaction()`, and `commitTransaction()` are dedicated to starting, aborting, and committing an atomic transaction, respectively. Some other methods provide additional data on atomic transactions, for example, the transaction depth, the memory used, and the memory still available in the transaction commit buffer.

Methods of the group `MakeTransient...Array` are used to create transient arrays of the types `boolean`, `byte`, `short`, and any other custom object type. Apart from specifying the number of elements in an array, these methods identify an event determining when the array elements are cleared (see Section 10.1.3). The method `isTransient` may be used to verify whether the given object is transient or not, and, if yes, to determine the type of event on which the content of the object is cleared to a default value.

Other class methods are rather specific, and their detailed description can be found in the Java Card API reference manuals.

11.1.2 `Applet` Class

The abstract class `Applet` must be implemented by any Java Card applet. The class inherits all necessary functionality for an applet's installation, registration, and execution. The core methods of the class were discussed in Section 8.3; Table 11.1 gives a brief summary of these core methods.

Two other class methods should also be mentioned. The purpose of the method `selectingApplet()` is quite interesting. It returns a `boolean` value indicating whether an applet has just been selected or not. The method is used in an applet `process` method in order to distinguish the applet SELECT APDU from any other SELECT APDUs that might be sent to the applet (see Figure 8.3).

Table 11.1
Summary of the Class `Applet` Core Methods

Method	Description
`install`	Called by JCRE in order to create an instance of an applet.
`register`	Invoked by an applet in order to register itself within JCRE.
`select()`	Called by JCRE to inform an applet that it was selected. Must return true to indicate a successful selection.
`deselect()`	Called by JCRE to inform an applet that either another applet was selected or the applet was selected again.
`process`	Called by JCRE in order to process an incoming APDU. The APDU itself is given as a method parameter.

The `getShareableInterfaceObject` method is called by JCRE in order to obtain a shareable interface object from this applet. For details related to object sharing between applets run on one card, see Section 10.2.2.

11.1.3 APDU Class

The `APDU` class encapsulates a complete set of features needed to process an incoming APDU, to prepare an outgoing response APDU, and to send it out. The `APDU` object is owned by JCRE. An applet receives its instance as a parameter of the applet `process` method. The `APDU` object has a byte array buffer that is used for storing both header and data bytes of incoming APDUs and data bytes of response APDUs.

We should mention that the class is dedicated only to command and response APDUs built in accordance with ISO/IEC 7816-4. Another convenient feature of the `APDU` class is that its methods apply to any (T = 0 or T = 1) communication protocol used by a smart card.

When the `APDU` object is passed by JCRE to the `process` method of an applet, its buffer array already contains the incoming APDU header, that is, the CLA, INS, P1, P2, and L_c bytes. The reference to the APDU byte array buffer can be obtained with the class method `getBuffer()`. To place data bytes, if any, in the buffer array, another method of the `APDU` class is called: the `setIncomingAndReceive()` method. A remarkable property of this method is that it places only as many bytes to the array buffer as can securely fit there, thereby avoiding buffer overflow.

If the incoming APDU contains more data bytes than will fit the array buffer (as it was copied by the `setIncomingAndReceive()` method), the remaining bytes can be placed to the buffer using subsequent calls of the `receiveBytes` method. This method also copies to the buffer only as many bytes as it will fit there.

The following example demonstrates the basics of reading APDU header and data bytes:

```
public void process(APDU apdu) {
    // get the reference to the array buffer
    byte[] buffer = apdu.getBuffer();
    // read CLA and INS bytes from the buffer
    byte cla  =  buffer[ISO7816.OFFSET_CLA];
    byte ins  =  buffer[ISO7816.OFFSET_INS];
    ...
    // incoming APDU has some data bytes; read them
    short bytesRead = apdu.setIncomingAndReceive();
    ...
}  // process(APDU apdu)
```

Abstract interface `ISO7816` contains static constants defining offsets of particular APDU bytes in the array buffer. Naturally, CLA, INS, P1, P2, and L_c have offsets 0, 1, 2, 3, and 4, respectively. The offset of the beginning of the data bytes is 5 and is defined by the static constant `OFFSET_CDATA` of the interface `ISO7816`.

Generating a response APDU is as easy as processing a command APDU. The class has a number of methods for preparing and sending out a response APDU. The easiest and most convenient method is `setOutgoingAndSend`. This method prepares a response APDU and sends it out immediately. It is used when all response APDU data fit into the array buffer. For instance, sending two bytes of response can be accomplished in the following way:

```
buffer[0] = (byte) 0xA0;
buffer[1] = (byte) 0xB0;
setOutgoingAndSend( (short) 0, (short) 2);
```

The first parameter of the method specifies an offset of the outgoing data in the array buffer (note that the same buffer is used for receiving and sending). The second parameter specifies the length of the outgoing data. As a result of the method execution, JCRE will send out a response APDU with two data bytes and the success status word 90 00 H automatically attached.

The same can be accomplished by using three other methods sequentially: `setOutgoing()`, `setOutgoingLength`, and `sendBytes`. To send large response APDUs, the method `sendBytesLong` is used.

11.1.4 `OwnerPIN` Class

The `OwnerPIN` class is an implementation of the `PIN` interface comprising functionality related to PIN verification (see Section 10.4). The class maintains the PIN value, a maximum number of unsuccessful tries allowed, and a try counter. If the maximum number of unsuccessful tries exceeds the defined limit, the `PIN` is *blocked*, meaning that even at this point presenting a correct PIN value will not result in successful PIN verification.

The class maintains the PIN validation flag indicating whether the PIN was successfully verified since the last reset of the card or of the `OwnerPIN` object. The validation flag is stored in the volatile memory of a card, which guarantees that its value is cleared after each card reset. The value of the validation flag is accessible via the method called `isValidated`.

Another class method `check` allows the PIN value to be verified against a given value. If the correct value is presented, the try counter is reset to its maximum value and the validation flag is set to true. The method returns a boolean value indicating whether the PIN verification was successful or not.

Other class methods make it possible to unblock the PIN (`resetAndUnblock`), to change the PIN value (`update`), to retrieve the remaining number of tries until the PIN will be blocked (`getTriesRemaining`), and to reset the PIN (`reset`).

11.1.5 `Util` Class

The `Util` class provides a number of handy utility functions that may be needed by an applet. Java Card specifications allow some of those functions to be implemented as native methods in order to increase their performance. As all methods of the class are static, the class does not need to be instantiated. Instead, the JCRE instance of the class can be used to invoke a desired method.

Basically, the class consists of two groups of methods. The first group contains several methods for operations on arrays: comparing two byte arrays (`arrayCompare`), copying byte arrays atomically (`arrayCopy`) and nonatomically (`arrayCopyNonAtomic`), and filling a byte array with a given byte value nonatomically (`arrayFillNonAtomic`).

The second group contains methods that deal with conversions between types `short` and `byte`. The methods make it possible to construct a `short` value from two `byte` values given separately or in a byte array, and to divide a `short` value into two `byte` values.

11.1.6 Interface `ISO7816`

Interface `ISO7816` contains a wide range of constants related to protocols and data structures defined in ISO/IEC 7816-3 and ISO/IEC 7816-4. In general, the constants fall into the following main groups:

- Constants defining an offset of particular bytes (e.g., CLA, INS, etc.) in the APDU buffer;
- Constants defining CLA and INS codes in accordance with ISO/IEC 7816-4;
- Response status word codes defined in accordance with ISO/IEC 7816-4.

Each of the constants can be accessed through the respective JCRE instance.

11.2 Existing Implementations

During the last few years, Java Card has met with good support from smart card manufacturers, and a number of Java Card implementations have appeared on the market. Apart from the standard Java Card API, each manufacturer also provides some extension API. This section presents three well-known Java Card implementations and discusses their special features. Table 11.2 at the end of the chapter presents an overview of the main characteristics of the Java Card implementations discussed in this section.

11.2.1 Giesecke & Devrient Sm@rtCafé

The Sm@rtCafé 1.1 card from the German company Giesecke & Devrient,[1] though based on the Java Card 2.1 specifications, is not fully compliant with Java Card 2.1. The card supports both T = 0 and T = 1 protocols that are selectable through protocol type selection (PTS). It is implemented on a single-chip microcontroller of the Siemens SLE66 family. The card has

1. http://www.gdm.de.

1,280 bytes of RAM, 32 Kbytes of ROM, and from 8 to 16 Kbytes of EEPROM for operating systems, applications, and data. The card chip is based on an 8-bit architecture and operates on the 7.5-MHz frequency.

The Sm@rtCafé 1.1 card exists in two variations: standard and crypt. The crypt version extends the basic Java Card functionality of the standard version by a comprehensive cryptographic API delivered as separate packages: `com.gieseckedevrient.javacardx.crypto` and `com.gieseckedevrient.javacardx.cryptox`.

Basically, the cryptographic API provides support of the following algorithms and services:

- DES and DES3 algorithms;
- RSA algorithm with the key length up to 1,024 bits;
- Secure hash algorithm SHA-1;
- External and mutual authentication services based on the DES algorithm;
- Digital signature service according to ISO/IEC 14888-3;
- Session key derivation service.

The functionality of each service and algorithm is encapsulated in a separate class. For instance, the `Authentication` class provides all necessary functionality for performing external or mutual card terminal authentication. The class methods make it possible to request challenge data from a card, to generate a session key, and to perform mutual or external authentication by simple methods invocation. In a similar manner, classes `Signer` and `Verifier` include the functionality needed to generate and verify, respectively, RSA digital signatures.

In addition, Sm@rtCafé 1.1 cryptographic API contains the `SecureRandom` class, which provides a source of *cryptographically* secure random numbers; that is, random numbers that cannot easily be guessed, predicted, and so on. This feature is very important for security of challenge-response authentication services or session key derivation services.

The installation program of the Sm@rtCafé 1.1 card, called the Main Loader, allows the definition of four security levels for loading an applet:

1. An applet is loaded in plaintext.
2. A loaded applet is digitally signed and the signature is verified by the Main Loader.

3. A loaded applet is encrypted.
4. A loaded applet is both digitally signed and encrypted.

The Main Loader can be configured in such a way that further reconfigurations are not allowed and the card application memory area cannot be cleared. This ensures that a loaded applet (or applets) cannot be deleted from the card or replaced.

The development environment that comes with the Sm@rtCafé 1.1 development toolkit makes it possible to perform complete simulation of the Sm@rtCafé 1.1 Java Card virtual machine and to perform applet bytecode-level debugging and tracing.

11.2.2 Gemplus GemXpresso 211

GemXpresso 211 is a family of smart card products from the Gemplus company.[2] GemXpresso 211 V2 is the currently available representative of this family and is the first Java Card technology smart card that complies with both Java Card 2.1 specifications and Visa Open Platform 2.0 specifications. The card supports both T = 0 and T = 1 protocols; the T = 1 protocol is available only after a warm reset of the card. The card is based on the 8-bit microcontroller and has 32 Kbytes of ROM, 32 Kbytes of EEPROM, and 2 Kbytes of RAM.

The security of GemXpresso 211 V2 and its components has been evaluated and certified by a number of international bodies. The card itself was certified by Visa with the highest security level 3. Security of the previous version of the card was evaluated according to the Common Criteria and received the assurance level EAL1. Security of the card cryptographic hardware components was evaluated according to the Common Criteria and received the assurance level EAL3.

Cryptographic support of the GemXpresso 211 V2 is limited only to DES and DES3 implementation. The RSA algorithm, secure hash algorithms, and digital signature algorithms are not supported. Also, the card provides no support for authentication services. In accordance with Java Card specifications, cryptographic functions are placed in the packages `javacard.security` and `javacardx.crypto`.

Implementation of the Visa Open Platform specification makes it possible to establish a secure communication channel between a GemXpresso

2. http://www.gemplus.com.

211 V2 card and a card terminal on the APDU level. A secure channel ensures APDU integrity and confidentiality as well as communication session authenticity. Visa Open Platform API is provided in the form of a separate package `visa.openplatform`.

We should mention that the next-generation card of the GemXpresso 211 family, GemXpresso 211/PK, will support the RSA algorithm and secure hash algorithms MD5 and SHA.

The GemXpresso 211 V2 toolkit is supplied with the GemXpresso RAD 211 development environment. It allows simulation of applet execution on the card and performance of debugging.

11.2.3 Schlumberger Cyberflex Access

The Schlumberger company was the first smart card manufacturer to release a smart card programmed in the Java programming language. The latest Java Card technology smart card from Schlumberger is Cyberflex Access.[3] Cyberflex Access is compliant with an earlier version of Java Card, namely, Java Card 2.0. The card supports only T = 0 protocol and has 16 Kbytes of EEPROM.

The cryptographic facilities of Cyberflex Access include the following:

- DES and DES3 algorithm implementation;
- RSA algorithm implementation with the key size up to 1,024 bits;
- External and internal card terminal authentication services;
- SHA-1 secure hash algorithm implementation.

The cryptographic API that covers the implementation of the algorithms mentioned above is located in the `javacardx.crypto` package.

An interesting feature of Cyberflex Access is that, in contrast to other Java Card technology smart cards, it supports the ISO/IEC 7816 file system. The files are accessed and managed via the Loader application of a Cyberflex Access card. The Loader application can be regarded as an extended variation of the Java Card installation program. Besides loading applets, the Loader also supports commands for file access and management and basic security mechanisms, such as card holder verification (CHV).

3. http://www.cyberflex.slb.com.

Cyberflex Access is supplied with an extension API delivered as a separate package, `javacardx.framework`. The extension API provides classes and methods for card-specific APDU processing (`CyberflexAPDU` class), operations related to files (`CyberflexFile` class), and native operating system calls (`CyberflexOS` class).

Table 11.2 summarizes the Java Card implementations just discussed.

Table 11.2
Summary of Java Card Implementations

	Sm@rtCafé	GemXpresso 211	Cyberflex Access
Manufacturer	Giesecke & Devrient	Gemplus	Schlumberger
Resources	1,280 bytes RAM, 32 Kbytes ROM, up to 16 Kbytes EEPROM	2 Kbytes RAM, 32 Kbytes ROM, 32 Kbytes EEPROM	16 Kbytes EEPROM
Supported protocols	T = 0, T = 1	T = 0, T = 1	T = 0
Java Card version	Java Card 2.1	Java Card 2.1	Java Card 2.0
Other specifications	—	Visa Open Platform 2.0	—
Cryptographic algorithms	DES, DES3, RSA SHA-1	DES, DES3	DES, DES3, RSA SHA-1
Security services	External and mutual authentication, ISO/IEC 14888-3 digital signature, session key derivation	—	External and internal authentication

INTERNET/COMPUTER SECURITY

Java Card for E-Payment Applications is a clear, comprehensive guide to smart cards, EMV-based credit and debit transactions, Java Card, and OpenCard Framework (OCF). This easy-to-understand resource provides in-depth coverage of important standards, open specifications, and critical security issues, including common threats and security mechanisms regarding the card and its connection interface. The book explains how to program a Java Card applet, an OCF card service, and a terminal application.

Professionals learn how OCF is structured and how it communicates with a smart card. From ISO 7816 smart card communication card readers and card terminals, to the deployment of Java Card technology, this unique resource offers hands-on assistance. What's more, the book presents an informative case study of the credit/debit application, offering a detailed road map of the application development process based on EMV specifications.

Vesna Hassler is a member of A-SIT (Secure Information Technology Center, Austria). She received her B.Sc. and M.Sc. in electrical engineering from Zagreb University and her Ph.D. from Graz University of Technology. Dr. Hassler is the author of *Security Fundamentals for E-Commerce* (Artech House, 2001).

Martin Manninger is a manager of smart card projects at Austria Card, Ltd. He received his M.Sc. in economics and informatics and his Ph.D. in electrical engineering from the Vienna University of Technology.

Mikhail Gordeev is an independent consultant in the areas of network security and smart cards. Dr. Gordeev received his M.Sc. in computer science from Perm State Technical University and his Ph.D. in electrical engineering and information technology from the Vienna University of Technology.

Christoph Müller currently works for Xsoft, Ltd. His recent research activity includes OCF and smart cards. He received an M.Sc. in informatics from the Vienna University of Technology.

ISBN 1-58053-291-8
90000>
9 781580 532914

Artech House Publishers BOSTON • LONDON
www.artechhouse.com