8

Java Card Basics

This chapter is dedicated to the basics of Java Card technology. It first explains the Java Card architecture and its remarkable features. Particular attention will be paid to differences between Java Card and Java technologies. The section concludes with the major steps of Java Card application development and explains the Java Card application programming interface (API).

8.1 Java Card Architecture

One of the main ideas that encouraged the development of Java Card technology was to make smart card applications portable across different platforms. The advantages of Java, such as platform independence and language-level security, were already well known and appreciated. Hence, a plan to bring the power of Java to the world of smart cards emerged and was implemented. The Java Card platform is formed by a combination of a customized subset of the Java programming language and a Java run-time environment dedicated to smart cards and other resource-constrained devices. Due to the fact that smart cards still have low performance and limited resources, it was necessary to customize the Java language, which was initially developed for the world of traditional computers. Customization of Java resulted in the omission of some features that are either impossible to implement on a smart card or irrelevant to smart card applications. However, some features were also added to Java Card to accommodate the specifics of a

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.

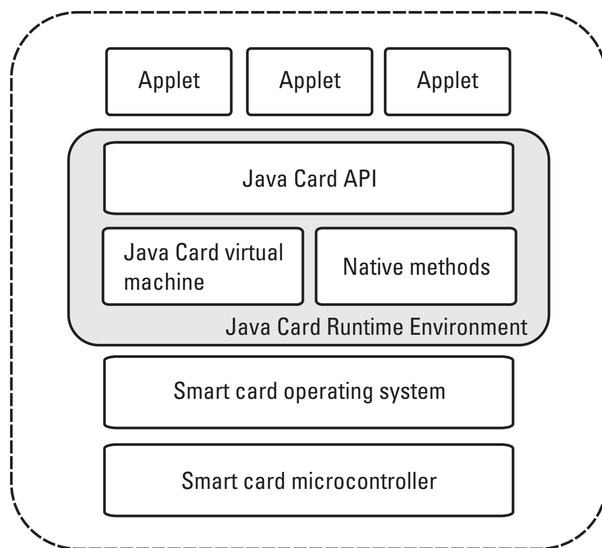

Figure 8.1 Java Card architecture.

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.

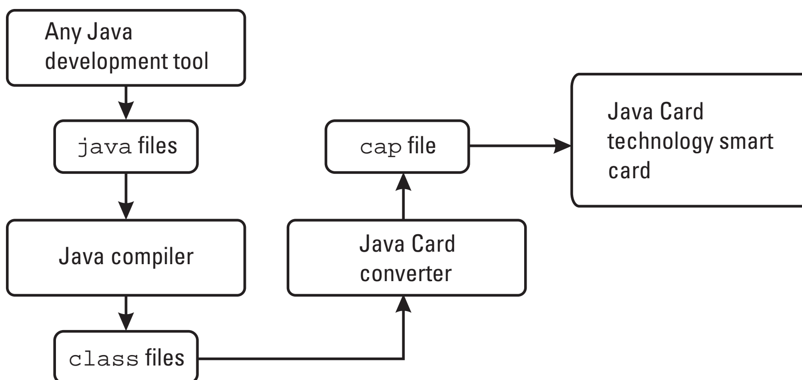

Figure 8.2 Java Card applet preparation.

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

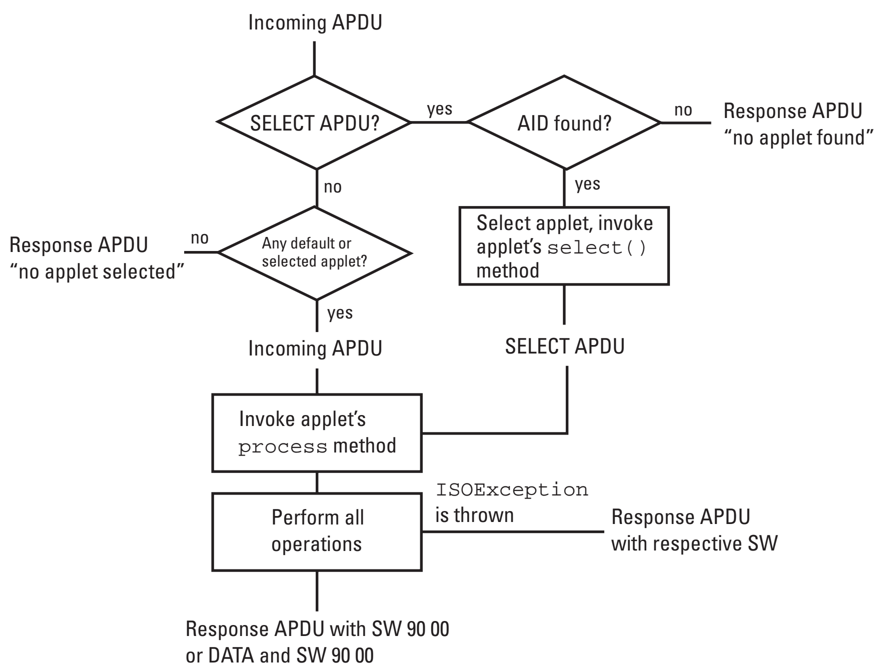

Figure 8.3 Command APDU processing by JCRE and an applet.

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.