CHAPTER 3

Java Card Technology Overview

Java Card technology enables programs written in the Java programming language to run on smart cards and other resource-constrained devices. This chapter gives an overview of Java Card technology—the system architecture and its components.

3.1 Architecture Overview

Smart cards represent one of the smallest computing platforms in use today. The memory configuration of a smart card might have on the order of 1K of RAM, 16K of EEPROM, and 24K of ROM. The greatest challenge of Java Card technology design is to fit Java system software in a smart card while conserving enough space for applications. The solution is to support only a subset of the features of the Java language and to apply a split model to implement the Java virtual machine.

The Java Card virtual machine is split into two part: one that runs off-card and the other that runs on-card. Many processing tasks that are not constrained to execute at runtime, such as class loading, bytecode verification, resolution and linking, and optimization, are dedicated to the virtual machine that is running off-card where resources are usually not a concern.

Smart cards differ from desktop computers in several ways. In addition to providing Java language support, Java Card technology defines a runtime environment that supports the smart card memory, communication, security, and application execution model. The Java Card runtime environment conforms to the smart card international standard ISO 7816.

The most significant feature of the Java Card runtime environment is that it provides a clear separation between the smart card system and the applications. The runtime environment encapsulates the underlying complexity and details of the smart card system. Applications request system services and resources through a well-defined high-level programming interface.

Therefore, Java Card technology essentially defines a platform on which applications written in the Java programming language can run in smart cards and other memory-constrained devices. (Applications written for the Java Card platform are referred to as *applets*.) Because of the split virtual machine architecture, this platform is distributed between the smart card and desktop environment in both space and time. It consists of three parts, each defined in a specification.

- The Java Card 2.1 Virtual Machine (JCVM) Specification defines a subset of the Java programming language and virtual machine definition suitable for smart card applications.
- The Java Card 2.1 Runtime Environment (JCRE) Specification precisely describes Java Card runtime behavior, including memory management, applet management, and other runtime features.
- The Java Card 2.1 Application Programming Interface (API) Specification describes the set of core and extension Java packages and classes for programming smart card applications.

3.2 Java Card Language Subset

Because of its small memory footprint, the Java Card platform supports only a carefully chosen, customized subset of the features of the Java language. This subset includes features that are well suited for writing programs for smart cards and other small devices while preserving the object-oriented capabilities of the Java programming language. Table 3.1 highlights some notable supported and unsupported Java language features.

It's no surprise that keywords of the unsupported features are also omitted from the language. Many advanced Java smart cards provide a garbage collection mechanism to enable object deletion.

Appendix A provides a comprehensive annotation of the Java Card language subset. For Java Card applets that require storing and manipulating big numbers, Chapter 14 provides programming tips for dealing with larger numbers without using large primitive data types.

Table 3.1 Supported and unsupported Java features

Supported Java Features	Unsupported Java Features
• Small primitive data types: `boolean`, `byte`, `short` • One-dimensional arrays • Java packages, classes, interfaces, and exceptions • Java object-oriented features: inheritance, virtual methods, overloading and dynamic object creation, access scope, and binding rules • The `int` keyword and 32-bit integer data type support are optional.	• Large primitive data types: `long`, `double`, `float` • Characters and strings • Multidimensional arrays • Dynamic class loading • Security manager • Garbage collection and finalization • Threads • Object serialization • Object cloning

3.3 Java Card Virtual Machine

A primary difference between the Java Card virtual machine (JCVM) and the Java virtual machine (JVM) is that the JCVM is implemented as two separate pieces, as depicted in Figure 3.1. The on-card portion of the Java Card virtual machine includes the Java Card bytecode *interpreter*. The Java Card *converter* runs on a PC or a workstation. The converter is the off-card piece of the virtual machine. Taken

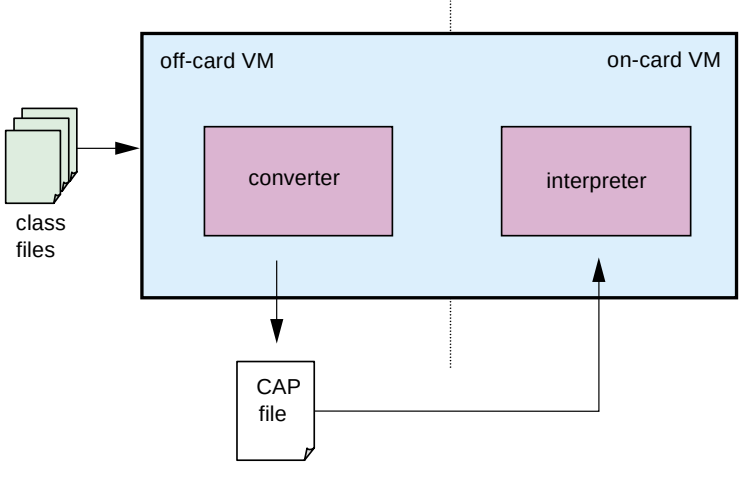

Figure 3.1 Java Card virtual machine

together, they implement all the virtual machine functions—loading Java class files and executing them with a particular set of semantics. The converter loads and preprocesses the class files that make up a Java package and outputs a CAP (converted applet) file. The CAP file is then loaded on a Java smart card and executed by the interpreter. In addition to creating a CAP file, the converter generates an export file representing the public APIs of the package being converted.

Java Card technology supports only a subset of the Java language. Correspondingly, the Java Card virtual machine supports only the features that are required by the language subset. Any unsupported language features used in an applet are detected by the converter.

3.3.1 CAP File and Export File

Java Card technology introduces two new binary file formats that enable platform-independent development, distribution, and execution of Java Card software. A CAP file contains an executable binary representation of the classes in a Java package. The JAR file format is used as the container format for CAP files. A CAP file is a JAR file that contains a set of components, each stored as an individual file in the JAR file. Each component describes an aspect of the CAP file contents, such as class information, executable bytecodes, linking information, verification information, and so forth. The CAP file format is optimized for a small footprint by using compact data structures and limited indirection. It defines a bytecode instruction set that is based on and optimized from the Java bytecode instruction set.

The “write once, run anywhere” quality of Java programs is perhaps the most significant feature of the Java platform. In Java technology, the class file is the central piece of the Java architecture. It defines the standard for the binary compatibility of the Java platform. Because of the distributed characteristic of the Java Card system architecture, the CAP file sets the standard file format for binary compatibility on the Java Card platform. The CAP file format is the form in which software is loaded onto Java smart cards. For example, CAP files enable dynamic loading of applet classes after the card has been made. That is how it gets the name converted applet (CAP) file.

Export files are not loaded onto smart cards and thus are not directly used by the interpreter. Rather, they are produced and consumed by the converter for verification and linking purposes. Export files can be thought of as the header files in the C programming language. An export file contains public API information for an entire package of classes. It defines the access scope and name of a class and

the access scope and signatures of the methods and fields of the class. An export file also contains linking information used for resolving interpackage references on the card.

The export file does not contain any implementation; that is, it does not contain bytecodes. So an export file can be freely distributed by an applet developer to the potential users of the applet without revealing the internal implementation details.

3.3.2 Java Card Converter

Unlike the Java virtual machine, which processes one class at a time, the conversion unit of the converter is a package. Class files are produced by a Java compiler from source code. Then, the converter preprocesses all the class files that make up a Java package and converts the package to a CAP file.

During the conversion, the converter performs tasks that a Java virtual machine in a desktop environment would perform at class-loading time:

- Verifies that the load images of the Java classes are well formed
- Checks for Java Card language subset violations
- Performs static variables initialization
- Resolves symbolic references to classes, methods, and fields into a more compact form that can be handled more efficiently on the card
- Optimizes bytecode by taking advantage of information obtained at class-loading and linking time
- Allocates storage and creates virtual machine data structures to represent classes

The converter takes as input not only the class files to be converted but also one or more export files. Besides producing a CAP file, the converter generates an export file for the converted package. Figure 3.2 demonstrates how a package is converted. The converter loads all the classes in a Java package. If the package imports classes from other packages, the converter also loads the export files of those packages. The outputs of the converter are a CAP file and an export file for the package being converted.

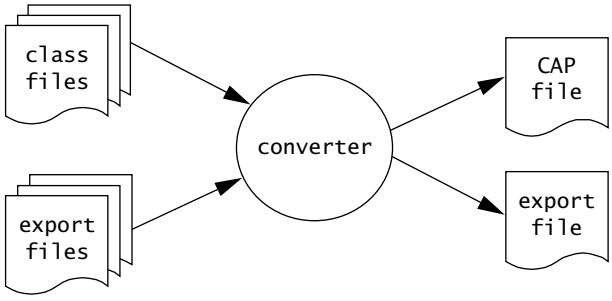

Figure 3.2 Converting a package

3.3.3 Java Card Interpreter

The Java Card interpreter provides runtime support of the Java language model and thus allows hardware independence of applet code. The interpreter performs the following tasks:

- Executes bytecode instructions and ultimately executes applets
- Controls memory allocation and object creation
- Plays a crucial role in ensuring runtime security

So far, the Java Card virtual machine has been described as comprising the converter and the interpreter. Informally, however, the Java Card virtual machine is defined as the on-card piece of the virtual machine—the interpreter, in our current definition. This convention has been applied in many early Java Card publications. Hence, for the remainder of this book, the terms Java Card interpreter and Java Card virtual machine are used synonymously unless otherwise stated. But readers should be aware that, when comparing the Java Card platform to the Java platform, the functions of executing Java class files are accomplished by the converter and the interpreter together.

3.4 Java Card Installer and Off-Card Installation Program

The Java Card interpreter does not itself load CAP files. It only executes the code found in the CAP file. In Java Card technology, the mechanisms to download and install a CAP file are embodied in a unit called the installer.

The Java Card installer resides within the card. It cooperates with an off-card installation program. The off-card installation program transmits the executable binary in a CAP file to the installer running on the card via a card acceptance device (CAD). The installer writes the binary into the smart card memory, links it with the other classes that have already been placed on the card, and creates and initializes any data structures that are used internally by the Java Card runtime environment. The installer and the installation program and how they relate to the rest of the Java Card platform are illustrated in Figure 3.3.

The division of functionality between the interpreter and the CAP file installer keeps the interpreter small and provides flexibility for installer implementations. More explanation of the installer is given in the coverage of the applet installation later in this chapter

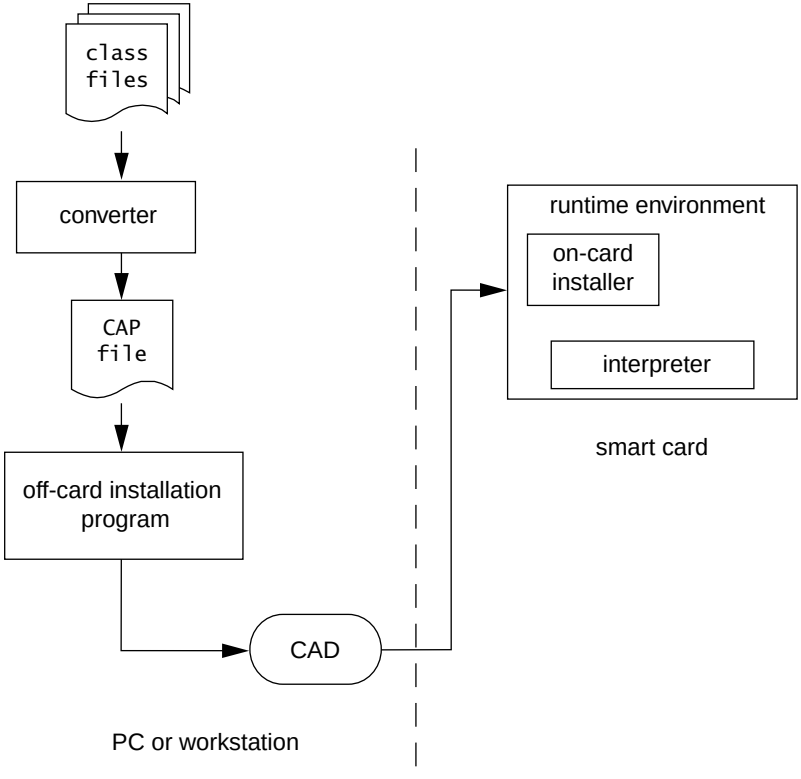

Figure 3.3 Java Card installer and off-card installation program

3.5 Java Card Runtime Environment

The Java Card runtime environment (JCRE) consists of Java Card system components that run inside a smart card. The JCRE is responsible for card resource management, network communications, applet execution, and on-card system and applet security. Thus, it essentially serves as the smart card's operating system.

As illustrated in Figure 3.4, the JCRE sits on top of the smart card hardware and native system. The JCRE consists of the Java Card virtual machine (the bytecode interpreter), the Java Card application framework classes (APIs), industry-specific extensions, and the JCRE system classes. The JCRE nicely separates applets from the proprietary technologies of smart card vendors and provides standard system and API interfaces for applets. As a result, applets are easier to write and are portable on various smart card architectures.

The bottom layer of the JCRE contains the Java Card virtual machine (JCVM) and native methods. The JCVM executes bytecodes, controls memory allocation, manages objects, and enforces the runtime security, as explained previously. The native methods provide support to the JCVM and the next-layer system classes. They are responsible for handling the low-level communication protocols, memory management, cryptographic support, and so forth.

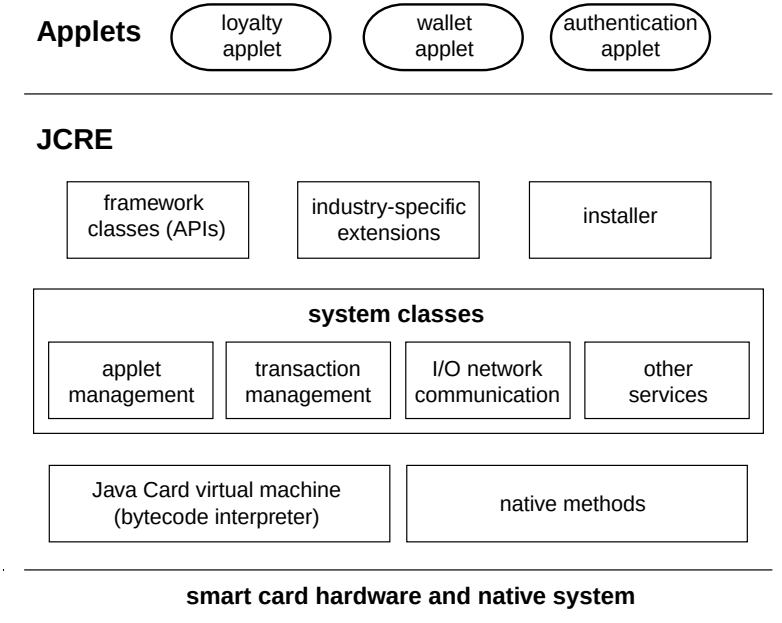

Figure 3.4 On-card system architecture

The system classes act as the JCRE executive. They are analogues to an operating system core. The system classes are in charge of managing transactions, managing communication between the host applications[1] and Java Card applets, and controlling applet creation, selection, and deselection. To complete tasks, the system classes typically invoke native methods.

The Java Card application framework defines the application programming interfaces. The framework consists of four core and extension API packages. The API classes are compact and customized for developing smart card applets. The major advantage of this framework is that it makes it relatively easy to create an applet. The applet developers can concentrate most of their effort on the details of the applets rather than on the details of the smart card system infrastructure. Applets access the JCRE services through API classes.

A specific industry or business can supply add-on libraries to provide additional services or to refine the security and system model. For example, the Open Platform extends the JCRE services to meet financial industries' specific security needs. Among many add-on features, it enforces issuers' control of the cards and specifies a standard set of commands for card personalization.

The installer enables the secure downloading of software and applets onto the card after the card is made and issued to the card holder. The installer cooperates with the off-card installation program. Together they accomplish the task of loading the binary contents of CAP files. The installer is an optional JCRE component. Without the installer, all card software, including applets, must be written into card's memory during the card manufacturing process.

Java Card applets are user applications on the Java Card platform. Applets are of course written in the subset of the Java programming language and controlled and managed by the JCRE. Applets are downloadable. Applets can be added to a Java smart card after it has been manufactured.

3.5.1 JCRE Lifetime

In a PC or a workstation, the Java virtual machine runs as an operating system process. Data and objects are created in RAM. When the OS process is terminated, the Java applications and their objects are automatically destroyed.

In a Java smart card, the Java Card virtual machine runs within the Java Card runtime environment. The JCRE is initialized at card initialization time. The JCRE initialization is performed only once during the card lifetime. During this process, the JCRE initializes the virtual machine and creates objects for providing

---

[1] Host applications are the applications running at the terminal side with which applets communicate.

the JCRE services and managing applets. As applets are installed, the JCRE creates applet instances, and applets create objects to store data.

Most of the information on a card must be preserved even when power is removed from the card. Persistent memory technology (such as EEPROM) is used to achieve this preservation. Data and objects are created in persistent memory. The lifetime of the JCRE is equivalent to the complete lifetime of the card. When power is removed, the virtual machine is only suspended. The state of the JCRE and the objects created on the card are preserved.

The next time the card is energized, the JCRE restarts virtual machine execution by loading data from persistent memory.[2] A subtle notion here is that the JCRE does not resume the virtual machine operation at the exact point where it lost power. The virtual machine is reset and executes from the beginning of the main loop. The JCRE reset differs from initialization, as it preserves applets and objects created on the card. During the reset, if a transaction was not previously completed, the JCRE performs any necessary cleanup to bring the JCRE into a consistent state.

3.5.2 How Does the JCRE Operate during a CAD Session?

The period from the time the card is inserted into the card acceptance device (CAD) and is powered up until the time the card is removed from the CAD is called a CAD session. During a CAD session, the JCRE operates like a typical smart card—it supports APDU I/O communication with a host application (Figure 3.5). APDUs

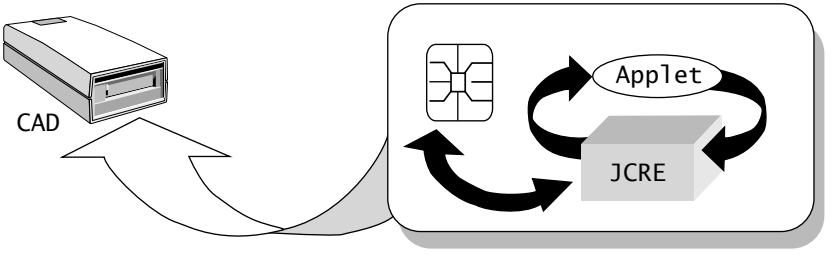

Figure 3.5 APDU I/O communication

---

[2] The JCRE also returns the answer to reset (ATR) to the host, indicating the card communication capabilities.

(application protocol data units) are data packets exchanged between applets and the host application. Each APDU contains either a command from the host to the applet or the response from the applet to the host

After a JCRE reset, the JCRE enters into a loop, waiting for APDU commands from the host. The host sends APDU commands to the Java Card platform, using the serial communication interface via the card input/output contact point.

When a command arrives, the JCRE either selects an applet to run as instructed in the command or forwards the command to the currently selected applet. The selected applet then takes control and processes the APDU command. When finished, the applet sends a response to the host application and surrenders control to the JCRE. This process repeats when the next command arrives. How applets process APDUs is explained further in Chapters 7 and 8.

3.5.3 Java Card Runtime Features

Besides supporting the Java language runtime model, the JCRE supports three additional runtime features:

- *Persistent and transient objects*—By default, Java Card objects are persistent and are created in persistent memory. The space and data of such objects span CAD sessions. For security and performance reasons, applets can create objects in RAM. Such objects are called transient objects. Transients objects contain temporary data that are not persistent across CAD sessions.
- *Atomic operations and transactions*—The Java Card virtual machine ensures that each write operation to a single field in an object or in a class is atomic. The updated field either gets the new value or is restored to the previous value. In addition, the JCRE provides transaction APIs. An applet can include several write operations in a transaction. Either all updates in a transaction are complete, or (if a failure occurs in the middle of the transaction) none of them proceeds.
- *Applet firewall and the sharing mechanisms*—The applet firewall isolates applets. Each applet runs within a designated space. The existence and operation of one applet has no effect on the other applets on the card. The applet firewall is enforced by the Java Card virtual machine as it executes bytecodes. In situations where applets need to share data or access JCRE services, the virtual machine permits such functions through secure sharing mechanisms.

3.6 Java Card APIs

The Java Card APIs consist of a set of customized classes for programming smart card applications according to the ISO 7816 model. The APIs contain three core packages and one extension package. The three core packages are `java.lang`, `javacard.framework`, and `javacard.security`. The extension package is `javacardx.crypto`.

Developers who are familiar with the Java platform will notice that many Java platform classes are not supported in the Java Card APIs. For example, the Java platform classes for GUI interfaces, network I/O, and desktop file system I/O are not supported. The reason is that smart cards do not have a display, and they use a different network protocol and file system structure. Also, many Java platform utility classes are not supported, to meet the strict memory requirements.

The classes in the Java Card APIs are compact and succinct. They include classes adapted from the Java platform for providing Java language support and cryptographic services. They also contain classes created especially for supporting the smart card ISO 7816 standard.

3.6.1 `java.lang` Package

The Java Card `java.lang` package is a strict subset of its counterpart `java.lang` package on the Java platform. The supported classes are `Object`, `Throwable`, and some virtual machine–related exception classes, as shown in Table 3.2. For the supported classes, many of the Java methods are not available. For example, the Java Card `Object` class defines only a default constructor and the `equals` method.

The `java.lang` package provides fundamental Java language support. The class `Object` defines a root for the Java Card class hierarchy, and the class `Throwable` provides a common ancestor for all exceptions. The supported exception classes ensure consistent semantics when an error occurs due to a Java language violation. For example, both the Java virtual machine and the Java Card virtual machine throw a `NullPointerException` when a null reference is accessed.

Table 3.2 Java Card `java.lang` package

`Object`	`Throwable`	`Exception`
`RuntimeException`	`ArithmeticException`	`ArrayIndexOutOfBoundsException`
`ArrayStoreException`	`ClassCastException`	`IndexOutOfBoundsException`
`NullPointerException`	`SecurityException`	`NegativeArraySizeException`

3.6.2 javacard.framework Package

The javacard.framework is an essential package. It provides framework classes and interfaces for the core functionality of a Java Card applet. Most important, it defines a base Applet class, which provides a framework for applet execution and interaction with the JCRE during the applet lifetime. Its role with respect to the JCRE is similar to that of the Java Applet class to a hosting browser. A user applet class must extend from the base Applet class and override methods in the Applet class to implement the applet's functionality.

Another important class in the javacard.framework package is the APDU class. APDUs are carried by the transmission protocol. The two standardized transmission protocols are T=0 and T=1. The APDU class is designed to be transmission protocol independent. In other words, it is carefully designed so that the intricacies of and differences between the T=0 and T=1 protocols are hidden from applet developers. Applet developers can handle APDU commands much more easily using the methods provided in the APDU class. Applets work correctly regardless of the underlying transmission protocol the platform supports. How to use the APDU class is explained in Chapter 8.

The Java platform class java.lang.System is not supported. The Java Card platform supplies the class javacard.framework.JCSystem, which provides an interface to system behavior. The JCSystem class includes a collection of methods to control applet execution, resource management, transaction management, and inter-applet object sharing on the Java Card platform.

Other classes supported in the javacard.framework package are PIN, utility, and exceptions. PIN is short for personal identification number. It is the most common form of password used in smart cards for authenticating card holders

3.6.3 javacard.security Package

The javacard.security package provides a framework for the cryptographic functions supported on the Java Card platform. Its design is based on the java.security package.

The javacard.security package defines a key factory class keyBuilder and various interfaces that represent cryptographic keys used in symmetric (DES) or asymmetric (DSA and RSA) algorithms. In addition, it supports the abstract base classes RandomData, Signature, and MessageDigest, which are used to generate random data and to compute message digests and signatures.

3.6.4 javacardx.crypto Package

The javacardx.crypto package is an extension package. It contains cryptographic classes and interfaces that are subject to United States export regulatory requirements. The javacardx.crypto package defines the abstract base class Cipher for supporting encryption and decryption functions.

The packages javacard.security and javacardx.crypto define API interfaces that applets call to request cryptographic services. However, they do not provide any implementation. A JCRE provider needs to supply classes that implement key interfaces and extend from the abstract classes RandomData, Signature, MessageDigest, and Cipher. Usually a separate coprocessor exists on smart cards to perform cryptographic computations. Chapter 10 explains how to support cryptographic functions in applets by using the classes in the javacard.security and javacardx.crypto packages.

3.7 Java Card Applets

Java Card applets should not be confused with Java applets just because they are all named applets. A Java Card applet is a Java program that adheres to a set of conventions that allow it to run within the Java Card runtime environment. A Java Card applet is not intended to run within a browser environment. The reason the name applet was chosen for Java Card applications is that Java Card applets can be loaded into the Java Card runtime environment after the card has been manufactured. That is, unlike applications in many embedded systems, applets do not need to be burned into the ROM during manufacture. Rather, they can be dynamically downloaded onto the card at a later time.

An applet class must extend from the javacard.framework.Applet class. The base Applet class is the superclass for all applets residing on a Java Card. The applet class is a blueprint that defines the variables and methods of an applet. A running applet on the card is an applet instance—an object of the applet class. As with any persistent objects, once created, an applet lives on the card forever.

The Java Card runtime environment supports a multiapplication environment. Multiple applets can coexist on a single Java smart card, and an applet can have multiple instances. For example, one wallet applet instance can be created for supporting the U.S. dollar, and another can be created for the British pound.

3.8 Package and Applet Naming Convention

Packages and programs that you are familiar with in the Java platform are uniquely identified using Unicode strings and a naming scheme based on Internet domain names. In the Java Card platform, however, each applet instance is uniquely identified and selected by an application identifier (AID). Also, each Java package is assigned an AID. When loaded on a card, a package is then linked with other packages on the card via their AIDs.

ISO 7816 specifies AIDs to be used for unique identification of card applications and certain kinds of files in card file systems. An AID is an array of bytes that can be interpreted as two distinct pieces, as shown in Figure 3.6. The first piece is a 5-byte value known as a RID (resource identifier). The second piece is a variable-length value known as a PIX (proprietary identifier extension). A PIX can be from 0 to 11 bytes in length. Thus an AID can range from 5 to 16 bytes in total length.

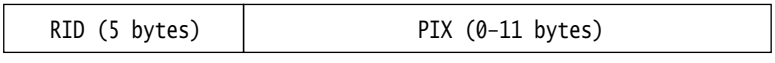

Figure 3.6 Application identifier (AID)

ISO controls the assignment of RIDs to companies; each company has a unique RID. Companies manage assignment of PIXs in AIDs. This section provides the brief description of AIDs. For complete details, refer to ISO 7816-5, AID Registration Category D format.

In the Java Card platform, the AID for a package is constructed by concatenating the company's RID and a PIX for that package. An applet AID is constructed similarly to a package AID. It is a concatenation of the applet provider's RID and the PIX for that applet. An applet AID must not have the same value as the AID of any package or the AID of any other applet. However, since the RID in an AID identifies an applet provider, the package AID and the AID(s) of applet(s) defined in the package must share the same RID.

The package AID and the default applet AID for each applet defined in the package are specified in the CAP file. They are supplied to the converter when the CAP file is generated.

3.9 Applet Development Process

Development of a Java Card applet begins as with any other Java program: a developer writes one or more Java classes and compiles the source code with a Java compiler, producing one or more class files. Figure 3.7 demonstrates the applet development process.

Next, the applet is run, tested, and debugged in a simulation environment. The simulator simulates the Java Card runtime environment on a PC or a workstation. In the simulation environment, the applet runs on a *Java virtual machine*, and thus the class files of the applet are executed. In this way the simulator can utilize

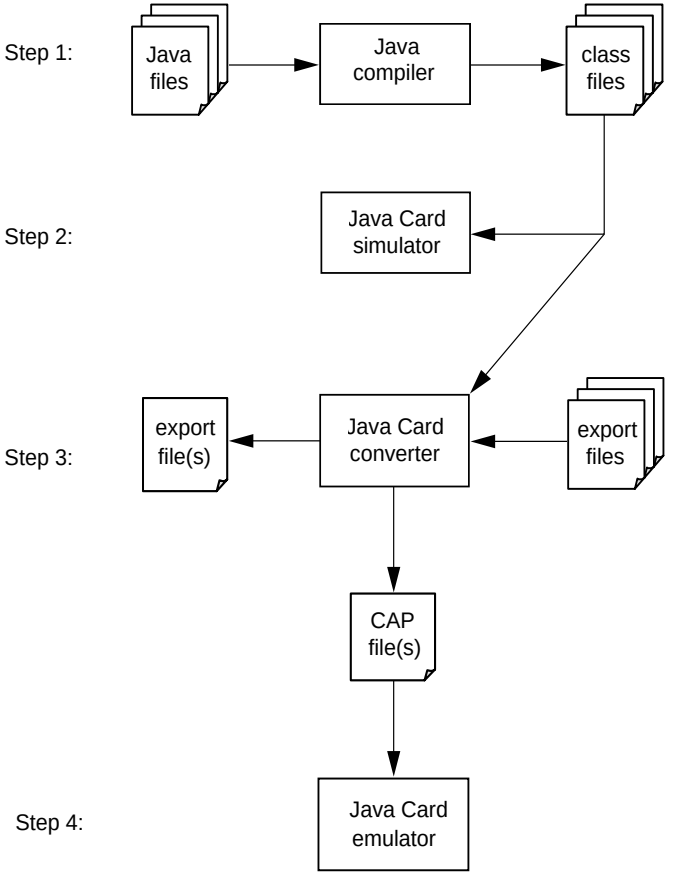

Figure 3.7 Applet development process

many Java development tools (the virtual machine, debugger, and other tools) and allow the developer to test the applet's behavior and quickly see the applet's results without going through the conversion process. During this step, the overall functional aspects of the applet are tested. However, some of the Java Card virtual machine runtime features, such as the applet firewall and the transient and persistent behavior of objects, cannot be examined.

Then the class files of the applet that make up a Java package are converted to a CAP file by using the Java Card converter. The Java Card converter takes as input not only the class files to be converted but also one or more export files. When the applet package is converted, the converter can also produce an export file for that package. A CAP file or an export file represents one Java package. If an applet comprises several packages, a CAP file and an export file are created for each package.

In the next step, the CAP file(s) that represent the applet are loaded and tested in an emulation environment. The emulator also simulates the Java Card runtime environment on a PC or a workstation. However, the emulator is a more sophisticated testing tool. It encompasses a *Java Card virtual machine* implementation. The behavior of the applet executing in the emulator should be the same as its behavior running in a real card. In this development phase, not only is the applet further tested, but also the runtime behavior of the applet is measured.

Most Java Card simulators and emulators come with a debugger. The debugger allows the developer to set breakpoints or single-step the program, watching the execution state of the applet change in the simulated or emulated Java Card runtime environment.

Finally, when the applet is tested and ready to be downloaded into a real card, the applet, represented by one or several CAP files, is loaded and installed in the Java smart card.

3.10 Applet Installation

When a Java smart card is manufactured, the smart card proprietary system and the Java Card runtime environment—including native methods, the Java Card virtual machine, API classes, and libraries—are burned into ROM. This process of writing the permanent components into the nonmutable memory of a chip is called *masking*. The technology for performing masking is a proprietary technology of a smart card vendor and is not discussed further in this book.

3.10.1 ROM Applets

Java Card applet classes can be masked in ROM together with the JCRE and other system components during the process of card manufacturing. Applet instances are instantiated in EEPROM by the JCRE during JCRE initialization or at a later stage. Such applets are called ROM applets.

The ROM applets are default applets that come with the card and are provided by card issuers. Because ROM applet contents are controlled by issuers, Java Card technology allows ROM applets to declare native methods whose implementations are written in another programming language, such as C or assembly code. Native methods are not subject to security checks enforced by the Java Card virtual machine.

3.10.2 Preissuance or Postissuance Applets

Alternatively, Java Card applet classes and associated class libraries can be downloaded and written into the mutable memory (such as EEPROM) of a Java smart card after the card is manufactured. Such applets can be further categorized as preissuance or postissuance applets. The terms preissuance and postissuance derive from the fact that applets are downloaded before or after the card has been issued. Preissuance applets are treated the same way as the ROM applets; both are controlled by the issuer.

Unlike ROM applets or preissuance applets, postissuance applets are not allowed to declare native methods. The reason is that the JCRE has no way to control the applet contents. Allowing downloaded applets to contain native code could compromise Java Card security.

The following subsections focus on postissuance applet installation. Usually preissuance applets are loaded using the same mechanism as postissuance applets, but Java Card technology leaves the decision to the card issuers.

3.10.3 Postissuance Applet Installation

Applet installation refers to the process of loading applet classes in a CAP file, combining them with the execution state of the Java Card runtime environment, and creating an applet instance to bring the applet into a selectable and execution state.

On the Java Card platform, the loading and installable unit is a CAP file. A CAP file consists of classes that make up a Java package. A minimal applet is a Java package with a single class derived from the class `javacard.framework.Applet`. A more complex applet with a number of classes can be organized into one Java package or a set of Java packages.

To load an applet, the off-card installer takes the CAP file and transforms it into a sequence of APDU commands, which carry the CAP file content. By exchanging the APDU commands with the off-card installation program, the on-card installer writes the CAP file content into the card's persistent memory and links the classes in the CAP file with other classes that reside on the card. The installer also creates and initializes any data that are used internally by the JCRE to support the applet. If the applet requires several packages to run, each CAP file is loaded on the card.

As the last step during applet installation, the installer creates an applet instance and registers the instance with the JCRE.[3] To do so, the installer invokes the `install` method:

```
public static void install(byte[] bArray, short offset, byte length)
```

The `install` method is an applet entry point method, similar to the `main` method in a Java application. An applet must implement the `install` method. In the `install` method, it calls the applet's constructor to create and initialize an applet instance. The parameter `bArray` of the `install` method supplies installation parameters for applet initialization. The installation parameters are sent to the card along with the CAP file. The applet developer defines the format and content of the installation parameters.

After the applet is initialized and registered with the JCRE, it can be selected and run. The JCRE identifies a running applet (an applet instance), using an AID. The applet can register itself with the JCRE by using the default AID found in the CAP file, or it can choose a different one. The installation parameters can be used to supply an alternative AID.

The `install` method can be called more than once to create multiple applet instances. Each applet instance is identified by a unique AID.

In the Java Card environment, an applet can be written and executed without knowing how its classes are loaded. An applet's sole responsibility during installation is to implement the `install` method.

3.10.4 Error Recovery during Applet Installation

The installation process is transactional. In case of an error, such as programmatic failure, running out of memory, card tear, or other errors, the installer discards the

---

[3] In a JCRE implementation, the operation for creating an applet instance can be performed at a later stage after applet installation.

CAP file and any applets it had created during installation and recovers the space and the previous state of the JCRE.

3.10.5 Installation Constraints

Readers should be aware that applet installation is different from dynamic class loading at runtime, which is supported on a Java virtual machine on the desktop environment. Java Card applet installation simply means to download classes through an installation process after the card has been made.

Therefore, Java Card applet installation has two finer points. First, applets executing on the card may refer only to classes that already exist on the card, since there is no way to download classes during the normal execution of applet code.

Second, the order of loading must guarantee that each newly loaded package references only packages that are already on the card. For example, to install an applet, the `javacard.framework` package must be present in the card, because all applet classes must extend from the class `javacard.framework.Applet`. An installation would fail if there were circularity such that package A and package B reference each other.