Java Card Applet Developer's Guide

Sun Microsystems, Inc
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300

Revision 1.10, July 17, 1998

Contents

Figures

Tables

Preface

Java™ Card™ technology combines a subset of the Java programming language with a runtime environment optimized for smart cards and related, small-memory embedded devices. The goal of Java Card technology is to bring many of the benefits of Java software programming to the resource-constrained world of smart cards.

This document demonstrates the concepts and APIs that developers need to write applications (applets) for the Java Card platform. This document is specific to version 2.0 of the Java Card API specification for use with version 1.0.2 of the Java Development Kit (JDK). After reading this guide, a developer will have enough knowledge of the Java Card technology programming concepts and the Java Card API to develop Java software applets for smart cards.

Who Should Use This Guide?

Java language developers, who wish to extend their development efforts onto smart card platforms, are the intended audience of this guide. It is also intended for use by existing smart card developers who are accustomed to programming in assembler or C.

Before You Read This Guide

Before reading this guide, you should familiarize yourself with the Java programming language, the Java Virtual Machine, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology is the Sun Microsystems, Inc. website, located at: http://java.sun.com. To download the JDK, see http://java.sun.com/products/jdk/.

How This Guide Is Organized

Chapter 1, “Overview,” provides an overview of smart cards and the Java Card technology architecture.

Chapter 2, “Java Card Technology,” provides a look at Java Card technology by way of comparison with Java technology.

Chapter 3, “Creating a Java Card Applet,” provides a sample Java Card applet and describes how to create your first Java Card applet.

Chapter 4, “Optimizing Java Card Applets,” describes the programming considerations for the resource-constrained environment of the smart card.

Chapter 5, “Files,” explains how to use Java Card technology’s file classes.

Chapter 6, “Cryptography,” explains how to use Java Card technology’s cryptography classes.

Glossary is a list of words and their definitions to assist you in using this guide.

Related Documents

References to various documents or products are made in this manual. You should have the following documents available:

- *Java Card 2.0 Application Programming Interface*, Sun Microsystems, Inc.
- *Java Card 2.0 Language Subset and Virtual Machine Specification*, Sun Microsystems, Inc.
- *The Java Language Specification* by James Gosling, Bill Joy, and Guy L. Steele. Addison-Wesley, 1996, ISBN 0-201-63451-1.
- *The Java Virtual Machine Specification* (Java Series) by Tim Lindholm and Frank Yellin. Addison-Wesley, 1996, ISBN 0-201-63452-X.
- ISO 7816 Specification Parts 1-6.

1. Overview

The Java Card specifications enable Java technology to run on smart cards and other devices with limited memory. To simplify the material, the focus in this document is on the smart card. A smart card is identical in size to a typical credit card and stores and processes information through the electronic circuits embedded in silicon in the plastic substrate of the card. There are two basic types of smart cards: memory and intelligent. A memory card stores data locally, but does not contain a CPU for performing computations on that data. An intelligent (smart) card includes a microprocessor and can perform calculations on locally-stored data.

There are several unique benefits of the Java Card technology in these smart cards, such as:

- **Platform Independent**—Java Card applets that comply with the Java Card API specification will run on cards developed using the Java Card Application Environment (JCAE), allowing developers to use the same Java Card applet to run on different vendors' cards.
- **Multi-Application Capable-**—Multiple applications can run on a single card. In the Java programming language, the inherent design around small, downloadable code elements makes it easy to securely run multiple applications on a single card.
- **Post-Issuance of Applications**—The installation of applications, after the card has been issued, provides card issuers with the ability to dynamically respond to their customer's changing needs.
- **Flexible**—The object-oriented methodology of the Java Card technology provides flexibility in programming smart cards.
- **Compatible with Existing Smart Card Standards**—The Java Card API is compatible with formal standards, such as, ISO7816, and industry-specific standards.

Smart Card Architecture

The smart card architecture consists of a communication interface, memory, and a CPU for performing calculations and processing information. The front and back of the card is pictured in FIGURE 1-1.

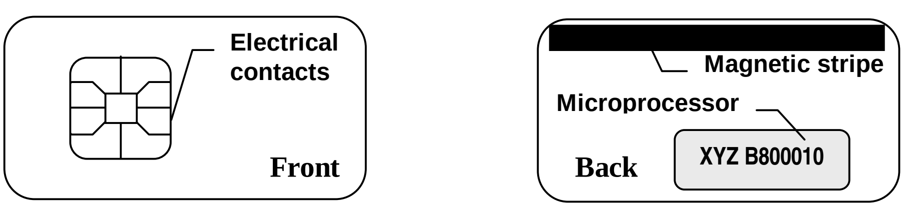

FIGURE 1-1 Front and Back of Smart Cards

Communication Interface

A smart card does not contain its own power supply, display, or keyboard. It interacts with a Card Acceptance Device (CAD) through using a communication interface, provided by a collection of eight electrical or optical contact points, as pictured in FIGURE 1-2.

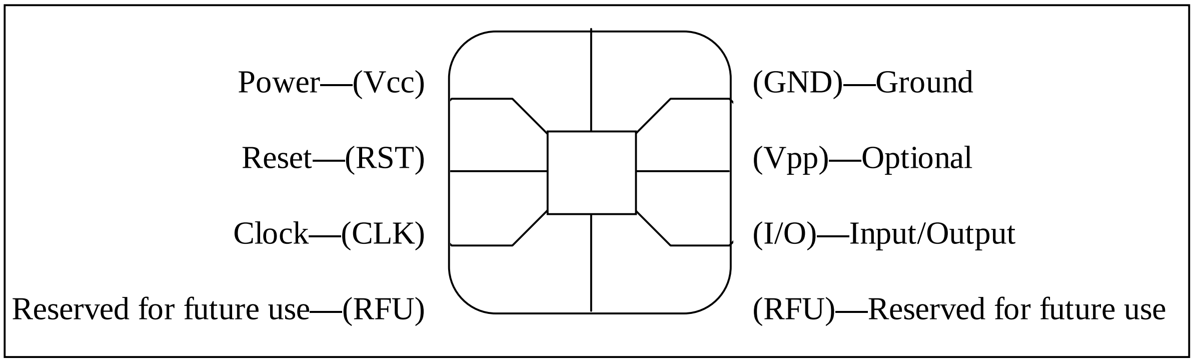

FIGURE 1-2 Eight Contact Points of the Smart Card Chip

Card Acceptance Device

The Card Acceptance Device (CAD) (also called a card reader, device reader, or card terminal) serves as a conduit for information into and out of the card. The card must be inserted into the CAD, as pictured in FIGURE 1-3, to provide the card with power (through its contacts, as described above).

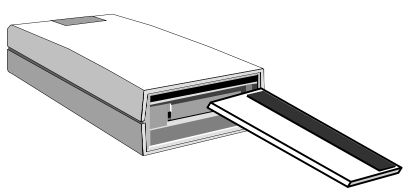

FIGURE 1-3 Card Acceptance Device

Smart Card CPU

The CPU on a smart card is simple. As a result of the cost sensitivity and low profile of smart card CPUs, they represent the low-end of the Java software platform market. Many Java language developers are accustomed to a world of powerful, multitasking CPUs with large amounts of RAM, virtual memory support, paging, and integrated I/O devices. Current smart card technology represents a return to the days when CPUs were 8-bit, single-tasking devices with 1KB of RAM or less.

On-Card Memory

There are three main types of memory on a smart card, they are:

- ROM (Read-Only Memory) — contains code and data that is read-only and cannot be modified. Information stored in ROM persists even after power to the card is disconnected.

- RAM (Random Access Memory) — is fast, volatile memory. Any information in RAM is lost when power to the card is disconnected. A typical Java Card platform implementation uses RAM for the frame and operand stacks and for storing temporary data.
- EEPROM (Electrical Erasable Programmable Read Only Memory) — is like ROM in that information in this type of memory persists across power sessions. However, EEPROM has the added advantage of being both readable and writeable, although writing to EEPROM is slower than writing to RAM and EEPROM is subject to wear. After a large number of writes to a particular byte, typically more than 100,000, the data integrity of that byte in EEPROM may fail. See the chip manufacturer's specification for details.

The applet developer should ensure that temporary fields that are updated frequently are components of transient arrays. This reduces potential wear on persistent memory and guarantees better write performance. As a rule of thumb, if a temporary field is being updated multiple times for every Application Protocol Data Unit (APDU), the applet developer should move it into a transient array.

Application Protocol Data Units

Smart cards communicate using a packet mechanism called Application Protocol Data Units (APDUs). Smart cards are reactive communicators—that is, they never initiate communications, they only respond to APDUs from the CAD. The communication model is command-response based—that is, the card receives a command APDU, performs the processing requested by the command, and returns a response APDU. See the *Creating a Java Card Applet* chapter for more information on working with APDUs.

The International Standards Organization (ISO) has set forth hardware and software specifications for creating inter-operable smart cards. These specifications are contained in the *ISO 7816 Parts 1-6* documents. For the purposes of developing Java Card applets, the most relevant document is *ISO 7816-4*.

Java Technology Smart Cards

A Java technology smart card is a smart card that can execute Java Card applets. These applets run in the Java Card environment, which may be as small as:

- 24K of ROM
- 16K of EEPROM
- 512 bytes of RAM

In addition to a CPU and memory, a Java technology smart card contains various software components, as described in the following table.

TABLE 1-1 Smart Card Software Components

Software Component	Description
Card OS	The Card Operating System (OS).

Native services	Performs the I/O, cryptographic, and memory allocation services of the card.
VM	The Java Card Virtual Machine (VM) provides bytecode execution and Java language support, including exception handling.
Framework	The set of classes which implement the API. This includes core and extension packages. Responsibilities include dispatching of APDUs, applet selection, managing atomicity, and installating applets.
API	The Application Programming Interface (API) defines the calling conventions by which an applet accesses the JCRE and native services.
JCRE	The Java Card runtime environment (JCRE) includes the Java Card Virtual Machine (VM), the framework, the associated native methods, and the API.
Industry extensions	Add-on classes that extend the applets installed on the card.
Applets	Programs written in the Java programming language for use on a smart card.

These components are illustrated in FIGURE 1-4.

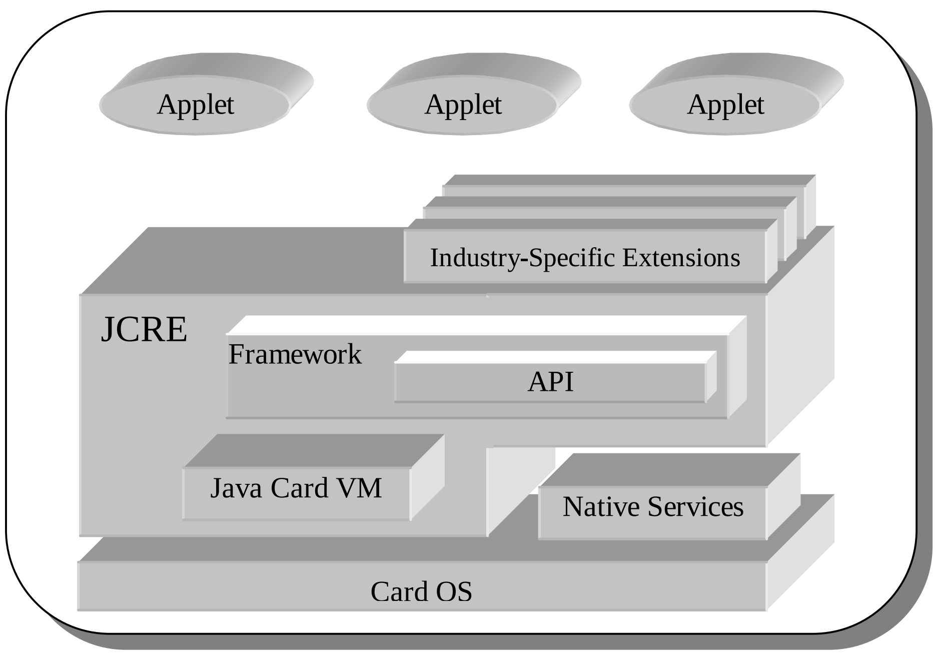

FIGURE 1-4 Java Card Technology Architecture

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. In effect, it is distributed in both space and time. The first piece of the VM executes off-card on a PC or workstation. This off-card part of the JCVM, the Java Card Converter, does all the work required for loading classes and resolving references where possible. The on-card part of the VM includes the bytecode interpreter.

The interface between the two pieces is a compiled applet (.cap file), as pictured in FIGURE 1-5, which is produced by the off-card VM during the development process and used by the on-card VM during the run process.

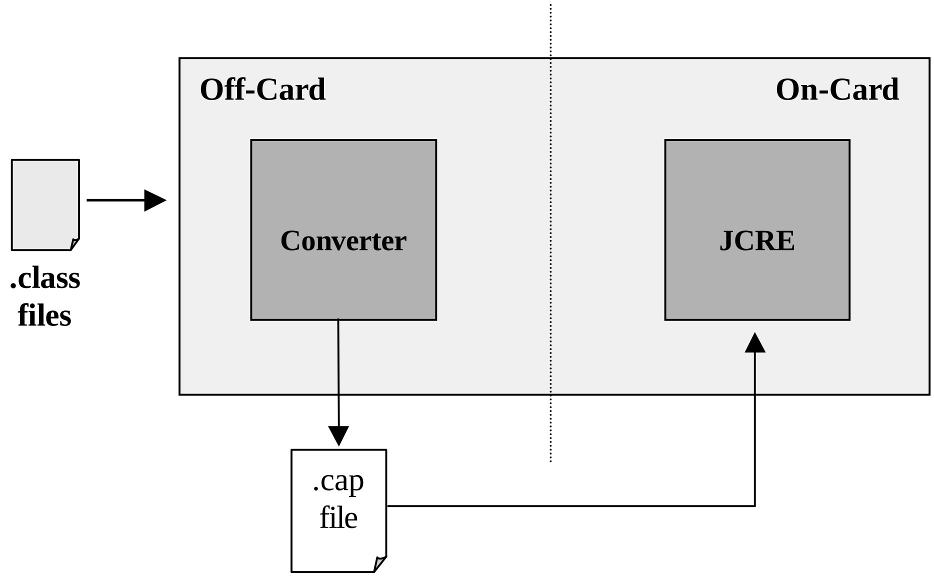

FIGURE 1-5 Creating a Java Card Applet

For further information, see the *Java Card Technology* chapter and the Java Card 2.0 Reference Implementation (JC2RI), available on the Sun Microsystems, Inc. website, located at http://java.sun.com.

Locked Card

There are a few conditions that cause the card to lock, preventing further use of the card. For example, a card might lock when an attempt to breach the card's security is detected (by perhaps, the PIN code being entered incorrectly more than five consecutive times). In this case, the issuer needs to be contacted (and the card may need to be returned) to reset the VM from such a locked state.

2. Java Card Technology

Java Card technology preserves many of the benefits of the Java programming language—productivity, security, robustness, tools, and portability—while enabling Java technology for use on smart cards. The Virtual Machine (VM), the language definition, and the core packages have been made more compact and succinct to bring Java technology to the resource-constrained environment of smart cards.

The Virtual Machine

The Java Card Virtual Machine (VM) provides **bytecode execution and Java language support, including exception handling**. The Java Card Runtime Environment (JCRE) includes a virtual machine (VM) and core classes to support APDU routing, ISO communication protocols, and transaction-based processing.

The Java Card VM is actually split into two parts, one for running off-card and the other for running on-card, as explained in the *Overview* chapter. The on-card Java Card VM executes bytecode, manages classes and objects, enforces separation between applications (firewalls), and enables secure data sharing.

The off-card Java Card VM contains a Java Card Converter tool for providing many of the verifications, preparations, optimizations, and resolutions that the Java VM performs at class-loading time. Dynamic class loading at runtime is not supported by the Java Card VM because:

- Dynamic class loading requires access to the storage location of the class file (refers to the disk or Internet) which is unavailable within a smart card environment
- Security aspects of the smart card environment prohibit most dynamic behavior (object dynamic binding is allowed)
- There are limited resources within the smart card environment

The Java Card Converter tool is an "early-binding" implementation of the Java VM. Every class referenced directly or indirectly by an applet must be bound into the applet's binary image when the applet is installed on the card. The Java Card Converter acts as an early-binding post-processor on the Java platform class files. The Java Card Converter performs the following steps:

1. **Verification**—checks that the load images of the classes are well formed, with proper symbol tables and checks for language violations, specific to the Java Card specifications

2. **Preparation**—allocates the storage for and creates the VM data structures to represent the classes, creates static fields and methods, and initializes static variables to default values

3. **Resolution**—replaces symbolic references to methods or variables with direct references, where possible

Performing these three steps in the Java Card Converter, before an applet is installed on the card, allows the on-card Java Card VM to be more compact and efficient. The following code fragment shows an example of where the Java Card Converter can resolve a method call before a class is ever installed on the smart card:

```
//FooBar() is declared in BaseType, overridden in ExtendedType.
BaseType b;
ExtendedType e = new ExtendedType();
b = e;
//e FooBar is called
(ExtendedType)b.FooBar();
```

Note – If the actual class of the object referenced by *b* is not ExtendedType at runtime (that is, we put a reference to some other extended type into *b* at runtime), then the above cast would cause a `ClassCastException`, just as it would on the Java platform.

Once an applet is installed on a Java Card-based smart card, it is considered loaded and ready to run (although some initializations and personalizations of the applet may be required). The JCRE then performs additional load-time initialization, which involves setting static constant initializers and initializing parameters declared with default values in interfaces.

Although the Java Card Converter performs as much early binding and resolution as possible, some late binding is also supported by the JCRE. The following code segment executes in the Java Card platform as it would on the Java platform:

```
//FooBar() is declared in BaseType, overridden in ExtendedType.
BaseType b;
ExtendedType e = new ExtendedType();
b = e;
//e FooBar is called
b.FooBar();
```

By late-binding the call to `b.FooBar()`, the JCRE guarantees that the method of the extended class `e` is called instead of the method of the base class for which the variable *b* is declared.

Late binding also allows Java Card technology to support virtual functions, for example:

```
//FooBar() is declared as an abstract method in BaseType and implemented
//by both ExtendedType1 and ExtendedType2
BaseType b;
ExtendedType1 e1 = new ExtendedType1();
ExtendedType2 e2 = new ExtendedType2();
b= e1;
//e1 FooBar is called
b.FooBar();
b= e2;
//e2 FooBar is called
b.FooBar();
```

Language Specifications

There are differences in the language specifications between the Java platform and the Java Card platform, resulting from the resource-constrained environment of the smart card. One main difference between the Java platform and the Java Card platform is that the Java Card platform supports only Java Card applets, not JDK-style applets or applications.

The Java Card API uses a subset of the Java programming language as defined in version 1.0.2 of the JDK. The reference implementation will run on any version of the JDK after and including version 1.1. The Java Card 2.0 Reference Implementation (JC2RI), available on the Sun Microsystems, Inc. website (http://java.sun.com), is based on the *Java Card 2.0 Language Subset and Virtual Machine Specification*.

The language differences between the Java platform and the Java Card platform are summarized in this section.

Threads

The Java Card platform does not support threads because current smart card central processing units (CPUs) cannot support efficient multitasking. As a result, none of the thread keywords are supported.

There is also no support in the Java Card platform for synchronized or volatile (used to control access to shared variables and methods among threads).

Garbage Collection

Java Card technology implementations are not required to support garbage collection, so the `finalize()` method is not supported.

Primitive Types

As in Java technology, Java Card technology supports the following primitive types: *byte*, *short*, and *boolean*. A byte is an 8-bit signed two's complement number with a possible range of values between -128 to 127. A short is a 16-bit signed two's complement number with a possible range of values between -32768 to 32767. Internally, Java Card technology represents the boolean type as a byte. This is in contrast to Java technology, which represents boolean internally as int. These are the only primitive types universally supported in Java Card technology, reflecting the 8-bit and 16-bit microprocessors on which Java Card technology currently executes.

The *int* type is available for use on some advanced 32-bit smart cards (the int type could actually be implemented on 16 or even 8 bit cards, but at a cost in execution and overhead). The int type represents a 32-bit signed two's complement number with a possible range of values between -2147483648 to 2147483647. The following table describes the supported primitive types.

TABLE 2-1 Supported Primitive Types

Type	Width	Range
byte	8 bits	-128, 127
short	16 bits	-32768, 32767
boolean	8 bits	TRUE or FALSE
int (supported on some platforms)	32 bits	-2147483648 to 2147483647

The Java Card platform does not support the *char*, *double*, *float*, or *long* primitive types. The *transient* and *volatile* declaration modifiers are unsupported. The following table describes the unsupported primitive types.

TABLE 2-2 Unsupported Primitive Types

Type	Width	Range
Long	64 bits	-2^{64}, $2^{64}-1$
Char	16 bits	Unicode v1.1.5 character set
Float	32 bit	Refer to IEEE 754
Double	64 bit	Refer to IEEE 754

Variables of type “byte” may be widened to “short” using the (short) cast. The widening occurs without loss of precision. Variables of type “short” may be narrowed to “byte” using the (byte) cast. The upper 8 bits of the short value are discarded. It is also possible to form a short from two byte values using the `Util.makeShort()` method found in the `javacard.framework` package.

To ensure that the results of arithmetic calculations are consistent with conventional Java technology, Java Card technology uses casting rules. The general rule is that the results of intermediate or unassigned arithmetic calculations must be explicitly cast to either a “byte” or short” value when used in combination with certain other operations (otherwise they would default to type “int”). An unassigned result is one which is not assigned to a variable (for example, an array index computed using an arithmetic calculation). In the following example, the calculation `(a+1)` yields an intermediate result which must be explicitly cast to either a “byte” or “short” value:

```
byte b;
short a;
byte array[] = new byte[10];
b = (byte)( (a+1)/2);           //This causes the Java Card Converter to
                                //issue an error
b = (byte)( (byte)(a+1)/2);     //Ok
b = (byte)( (short)(a+1)/2);    //Ok
b = array[a+1];                 //Error
b = array[(byte)(a+1)];         //Ok
b = array[(short)(a+1)];        //Ok
```

Replacing the intermediate calculation of `(a+1)` with the symbol *I* reduces the equation to:

```
b = (byte)(I/2);
```

A second arithmetic calculation `(I/2)` now occurs, but this second arithmetic calculation is neither intermediate nor unassigned. The result of the second arithmetic calculation is directly assigned to the variable *b*, and Java programming language rules require an explicit cast to the variable type, in this case a byte. Java Card technology is consistent with Java technology by requiring the explicit cast when the results of the arithmetic calculation are assigned to a typed variable.

The Java Card technology casting requirements on intermediate results of arithmetic calculations are necessary because of the manner in which Java technology handles the calculation of intermediate results. Java technology converts the intermediate results of operations involving bytes and shorts to int (32 bit) values, unless an explicit cast to short or byte is used. Because Java Card technology is designed to run on 8- and 16-bit CPUs, storing intermediate results in 32 bits is not possible. (See the Java Card Converter documentation for error reporting when an explicit cast is not performed.)

It can be shown that the overall result of a complex arithmetic expression (one involving intermediate results) may vary, depending on the number of bits used to store the intermediate results. This discrepancy is due to the fact that overflow bits, which result from the intermediate calculation, are **preserved** when more bits are used to store the intermediate values and are **truncated** when fewer bits are used. Explicitly casting the results of intermediate or unassigned calculations, explicitly defines the number of bits used to store the intermediate results, and ensures results consistent with Java technology.

Intermediate or unassigned arithmetic operations that are subject to the explicit casting rules are:

List #1

- Addition (+, ++)
- Subtraction (-, --)
- Unary negation (-)
- Left shift (<<)
- Multiplication (*)
- Division (/)

Intermediate arithmetic results of the operations listed above must be cast when used as the operands to any of the operations listed below:

List #2

- Remainder (%) (both operands)
- Unary negation (-)
- Right shift (>>) (operand being shifted)
- Unsigned right shift (>>>) (operand being shifted)
- Division (/) (both operands)

In addition, unassigned (as opposed to intermediate) arithmetic results must be explicitly cast when used in the following circumstances:

List #3

- As the element count when creating a new array
- As an array index
- As the condition of `if` expressions
- As the parameter to a method (also required with Java technology)
- As the decision value in `switch` statements

Intermediate results from operations in List #1, which are used as inputs to the bitwise AND operation (&), the bitwise OR operation (|), and the bitwise XOR operation (^), are subject to the casting rules if the result of the &, |, or ^ operation is itself an input to one of the operations in List #2 or List #3.

Here are some examples of applying the explicit casting rules:

```
short s=2, c=4, d=5;
byte a=1, b=3;
byte array[] = new byte[(byte)(a*b)];
s = (short) ( (byte)(a/b) >> 2);
s = (short) ( b >> (a*b));              //no cast required – intermediate
                                        //is not operand being shifted
s = (short) ((byte) (a*b)/c);
s = (short) ((short) (a*b)/(short)(c*d));
s = (short) array[(short)(a/b)];
s = (short) ((short)((c*c) & d)/a);     //cast required on result of &
                                        //operation
if ((short)(a*b*c) < 10) { … }
```

```
switch ((byte) (c >> b)) { … }
```

Arrays

Java Card technology supports single-dimensional arrays, but does not support multidimensional arrays because of the resource constraints of the smart card environment. Arrays must have elements which are one of the supported primitive types or objects. Array elements can be references to other arrays. The following are examples of valid array declarations for both the Java platform and the Java Card platform:

```
byte a[] = new byte[10];
byte a[] = {1,1,2};
static final short MAX_ARRAY = 15;
short s[] = new short[MAX_ARRAY];
AID aid[] = new AID[5];          //array of 5 AID object references
```

The following are examples of invalid array declarations:

```
byte b[][] = new byte[5][2];     //multidimensional arrays in all these
                                 //examples are not supported
short s[][] = new short[10][];
byte b[][];
```

Arrays are objects (just as they are in Java technology), so that the `Object` methods are available when using arrays. Two array references can be compared for equality using the `equals()` method:

```
if (c.equals(d)) {...}
```

The return value is always TRUE or FALSE—either the array references are equal or they are not. More elaborate comparisons between the elements of two byte-arrays can be performed using the `Util.arrayCompare()` method. This method allows you to compare any range of the elements of two arrays, with a return value indicating whether the range is less than, equal to, or greater than the other. Any range of one byte-array may be copied to the range of another (including the same array) using `Util.arrayCopy()`.

The `arrayCopy()` method ensures the atomicity of the copy operation—removing (tearing) the card in the middle of the operation does not result in a partially copied array. The non-atomic version of the same method is `Util.arrayCopyNonAtomic()`. A similar method, `Util.arrayFillNonAtomic()`, non-atomically fills the elements of a byte array with a specified value.

The memory implications of source-to-source array copies and atomic array operations are discussed in more detail in the section on *Optimizing Java Card Applets*.

Two consecutive byte array elements may be returned as a short value using `Util.getShort`. Likewise, two consecutive byte array elements may be set using the first and second bytes in a short value using `Util.setShort`.

Inheritance

The Java Card platform fully supports all of the inheritance features available in the Java platform. Method overrides, abstract methods, and interfaces are all supported. The `super` and `this` keywords are supported, with identical usage rules: both may be used only in an instance method, constructor, or in the initializer of an instance variable of a class.

Security

All of the Java language and compilation security features are supported by Java Card technology, such as bytecode verification, confirming that all targets in the applet are reachable, and public, private, and protected access modifiers.

Just as in the Java platform, members declared as:

- *Public* are accessible to other installed classes
- *Private* are accessible only from within the class in which it is declared
- *Protected* are accessible to and inherited by subclasses and accessible by code in the same package

---

Note – By default, members are accessible by any code within the same package.

Portability

The Java Card API allows applications written for one Java Card-based smart card platform, to run on any other Java Card-based smart card platform—allowing developers to use the same Java Card applet to run on different vendors' cards.

A Java Card applet will not execute as a Java applet does in a JDK runtime environment and a Java applet will not execute as a Java Card applet does in the JCRE. However, if the JCRE is simulated in the JDK runtime environment, the card applet will execute.

In general, Java Card technology provides your software with independence from the:

- CPU-specific features of the card (for example, instruction set, byte-ordering conventions, or data and instruction bus width)
- ISO protocol used to communicate between the card and the Card Acceptance Device

Java Card technology does not provide independence from all of the features of different models of smart cards. For example, some smart cards may support the int primitive data type or garbage collection—if your Java Card applet assumes such support is present, it will only work on smart cards that implement those features.

Exceptions

Java Card applets must be tested thoroughly to avoid any fatal errors during execution on the smart card. To assist developers in debugging the Java Card applets, the Java Card platform supports all of the Java programming language constructs for exceptions. You can include `try()`, `catch()`, or `finally()` constructs in your Java Card applets, and they work the same as in the Java platform. The `throw` keyword is also supported, however as discussed below, the preferred way to throw exceptions does not directly involve the use of `throw`.

The Java Card platform does not support all of the exception types found in the Java technology core packages because many of them are not applicable in a smart card context. For example, threads are not supported within the Java Card platform (as discussed in the *Language Specifications* section of this chapter) and as a result, none of the thread-related exceptions are supported. The following table lists all of the supported exceptions.

TABLE 2-3 Supported Exceptions

Exception	Package	Superclass	Type
Exception	java.lang	Throwable	Checked
RuntimeException	java.lang	Exception	Unchecked
ArithmeticException	java.lang	RuntimeException	Unchecked
ArrayStore	java.lang	RuntimeException	Unchecked
ArrayIndexOutOfBoundsException	java.lang	RuntimeException	Unchecked
ClassCastException	java.lang	RuntimeException	Unchecked
IndexOutOfBoundsException	java.lang	RuntimeException	Unchecked
NegativeArraySizeException	java.lang	RuntimeException	Unchecked
NullPointerException	java.lang	RuntimeException	Unchecked
SecurityException	java.lang	RuntimeException	Unchecked
APDUException	javacard.framework	RuntimeException	Unchecked
ISOException	javacard.framework	RuntimeException	Unchecked
PINException	javacard.framework	RuntimeException	Unchecked
SystemException	javacard.framework	RuntimeException	Unchecked
TransactionException	javacard.framework	RuntimeException	Unchecked
UserException	javacard.framework	Exception	Checked

To optimize memory usage, all exception objects can be pre-created at initialization time and their references saved permanently. When the exception event occurs, rather than create a new exception object, the code can:

1. Retrieve and reuse the reference for the desired exception object.
2. Fill in the reason code in the object.
3. Throw the object.

The JCRE pre-creates an instance of each kind of specific exception defined in the Java Card API. Most of these are unchecked exceptions. When these exception objects are needed, use the static method `throwIt()`. See *The Throwable Class* section for more information.

You can define your own exceptions by creating subclasses of class `Exception`. These are always checked exceptions. These exceptions can be thrown and caught as desired by the applet. However, during initialization the applet needs to:

1. Create a single instance of each such exception.
2. Save the reference in some persistent object field.
3. Reuse that instance whenever it is necessary to throw that exception.

Core Classes

The core classes of the Java Card API are more compact and succinct than those in the Java platform and provide essential services to Java Card applets. Developers familiar with the Java platform may not recognize some of the Java Card platform core classes, with the exception of `Object` and `Throwable` in `java.lang`.

The `String` classes, `I/O` classes, `AWT` classes, and `net` classes are not supported in the Java Card platform core classes because of memory constraints within the smart card environment.

The following table lists all of the Java Card platform packages and the classes they contain:

TABLE 2-4 Java Card Platform Core Classes

Package	Class
Core Classes	
java.lang	Object
.	Throwable
.	Various language exceptions
javacard.framework	AID
.	APDU
.	Applet
.	ISO
.	PIN
.	System
.	Util
.	Various related exceptions
Extension Classes	
javacardx.framework	File system classes
javacardx.crypto	Public key classes
.	Private key classes
.	Random number generator
.	Message digest
javacardx.cryptoenc	DES encryption classes

The Throwable Class

To support exceptions, the `Throwable` class is implemented within the Java Card platform. The implementation is designed such that exception objects are allocated once system-wide, then customized and reused by all installed classes. Although this is an option for handling exception objects, it is highly recommended as a space saver because the Java Card platform does not provide universal support for garbage collection or for explicit deallocation of allocated memory (providing support for explicit deallocation of allocated memory would violate the Java technology programming model).

Once an object is instantiated, the storage set aside for that object is reserved for the lifetime of the smart card—allowing a guarantee of space availability during future execution of that object.

By creating shared, system-wide exception objects, the Java Card platform allows exceptions to be thrown using a reused exception object. An applet simply retrieves a reference to a shared exception-object of the desired type, customizes the exception with an error code, and throws the exception by invoking the exception's `throwIt()` method—**not** by using the `throw` keyword.

Like the Java platform, all exceptions are derived from the `Throwable` class, for example:

```
public class Throwable
//Methods:
      public short getReason(); //get the error code to throw with the
                                //exception
      public void setReason();  //set the error code to throw with the
                                //exception
```

Unlike in the Java platform, the following methods are not part of `Throwable` because of memory constraints within the smart card environment:

```
toString
getMessage
getLocalizedMessage
fillInStackTrace
printStackTrace
```
(this method is not supported because there is no printer on a smart card)

The declaration of `Exception` in the Java Card platform is:

```
public class Exception extends Throwable
//Member variables:
      private static Exception exceptionInstance;
Methods:
public static void throwIt (short reason) throws Exception {
      if (exceptionInstance == null)
            exceptionInstance = new Exception();
      exceptionInstance.setReason (reason);
      throw exceptionInstance;
}
```

All Java Card platform exceptions in the core packages derive from `Exception` and override the `throwIt()` method. An example of throwing an exception using `throwIt()` is provided in the *Optimizing Java Card Applets* chapter.

The Object Class

The definition of the `Object` class in the Java Card platform is more succinct than within Java technology to accommodate the resource constraints of the smart card environment. The definition within the Java Card platform is as follows:

```
public class Object {
      public boolean equals(Object obj){ return false; }
}
```

The only supported method is `equals()`, which, when implemented by subclasses of `Object`, provides a comparison of the equality of two object references. Notice that there is no `getClass()` method implemented—`Class` objects are unreachable by an applet in the Java Card platform.

Other `Object` methods in the Java platform that are not in the Java Card platform include:

```
toString
hashcode
clone
wait
notify
notifyAll
finalize
```

As a result of the resource constraints in devices with limited memory, the following methods are not supported within the Java Card platform:

- The `hashcode()` method—hash tables are not supported
- The `clone()` method—objects cannot be cloned
- The `wait()`, `notify()`, and `notifyAll()` methods—threads are not supported
- The `System.exit()` method—the VM has a continuous life on an active card so there is no need for a method that instructs the JCRE to shut down the VM
- The `finalize()` method—garbage collection is not universally supported and the concept of relinquishing system resources owned by the object being garbage collected (for example, graphics) is not meaningful in the smart card environment

3. Creating a Java Card Applet

Off-the-shelf Java technology development tools can be used to create and compile the source code for Java Card applets. Once the Java software source code is compiled into a class file, a post-processed version of the applet, suitable for loading on Java technology smart cards, is created using the Java Card Converter tool, as discussed in the *Overview* chapter and pictured in FIGURE 3-1.

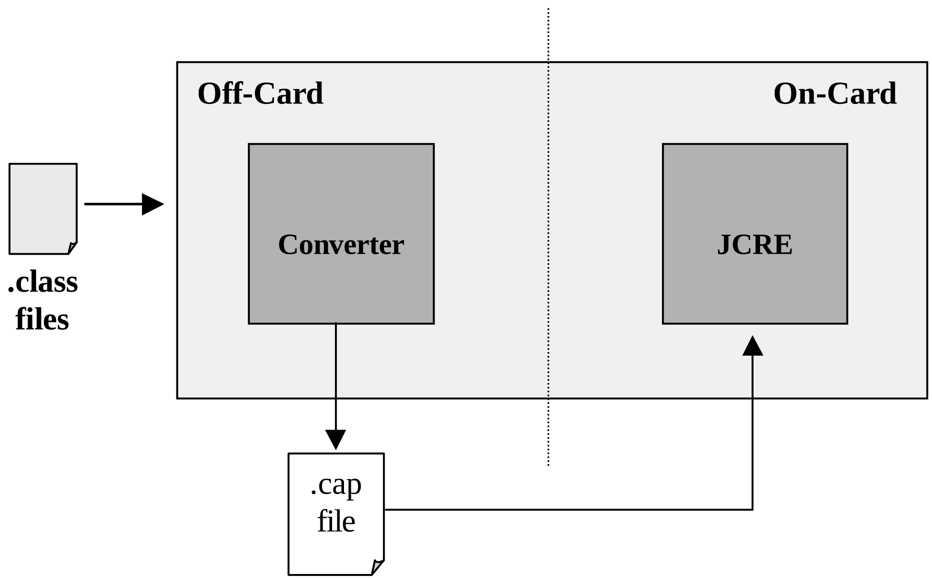

FIGURE 3-1 Creating a Java Card Applet

Java Card technology differs from Java technology as a result of the resource constraints of the smart card environment, as discussed in the *Java Card Technology* chapter. Smart cards communicate using a packet mechanism called APDUs, as discussed in the *Overview* chapter and in the *Working with APDUs* section of this chapter.

You now have enough background to begin the process of creating the source code for a Java Card applet. The source code for the applet begins with the same package designations found in the Java language (the `java.lang` package need not be expressly imported into your applet).

A Basic Example

The following code listing shows a basic example of a Java Card applet that can be post-processed by the Java Card Converter and installed onto a Java technology smart card:

package bank.purse;	Java Card supports package and identifier name convention as in standard Java technology.
import javacard.framework.*; import javacardx.framework.*;	
public class Wallet extends Applet { /* constants declaration */	An applet is an instance of a class which extends from: javacard.framework.Applet.
// code of CLA byte in the command APDU header final static byte Wallet_CLA =(byte)0xB0;	CLA identifies the application

// codes of INS byte in the command APDU header
final static byte Deposit = (byte) 0x10;
final static byte Debit = (byte) 0x20;
final static byte Balance = (byte) 0x30;
final static byte Validate = (byte) 0x40;
	INS specifies the application instructions.

// maximum number of incorrect tries before the
// PIN is blocked
final static byte PinTryLimit =(byte)0x03;
// maximum size PIN
final static byte MaxPinSize =(byte)0x04;
	PIN object parameters.

// status word (SW1-SW2) to signal that the
// balance becomes negative;
final static short SW_NEGATIVE_BALANCE = (short)
0x6910;
	Applet-specific static word.

/* instance variables declaration */ OwnerPIN pin; byte balance; byte buffer[]; // APDU buffer	

private Wallet() {
// It is good programming practice to allocate
// all the memory that an applet needs during
// its lifetime inside the constructor
pin = new OwnerPIN(PinTryLimit, MaxPinSize);
balance = 0;
register();
 	
} // end of the constructor
	private constructor—an instance of class Wallet is instantiated by its install method.
	
	The applet registers itself with the JCRE by calling `register` method, which is defined in class `Applet`.

public static void install(APDU apdu){
// create a Wallet applet instance
new Wallet();
 	
} // end of install method
	Method `install` is invoked by the JCRE as the last step in the applet installation process.

public boolean select() {
 	
// reset validation flag in the PIN object to
// false
pin.reset();
 	
// returns true to JCRE to indicate that the
// applet is ready to accept incoming APDUs.
return true;
 	
}// end of select method
	This method is called by the JCRE to indicate that this applet has been selected. It performs necessary initialization which is required to process the subsequent APDU messages.

`public void process(APDU apdu) {`
 	
`// APDU object carries a byte array (buffer) to`
`// transfer incoming and outgoing APDU header`
`// and data bytes between card and CAD`
`buffer = apdu.getBuffer();`
	After the applet is successfully selected, the JCRE dispatches incoming APDUs to this method.
	
	APDU object is owned and maintained by the JCRE. It encapsulates details of the underlying transmission protocol (T0 or T1 as specified in ISO 7816-3) by providing a common interface.

`// verify that if the applet can accept this`
`// APDU message`
`if (buffer[ISO.OFFSET_CLA] !== Wallet_CLA)`
`  ISOException.throwIt`
`    (ISO.SW_CLA_NOT_SUPPORTED);`
	When an error occurs, the applet may decide to terminate the process and throw an exception containing the status word (SW1 SW2) to indicate the processing state of the card.
	
	An exception that is not caught by an applet is caught by the JCRE.

`switch (buffer[ISO.OFFSET_INS]) {`
`  case Balance:   getBalance(apdu); return;`
`  case Debit:   debit(apdu); return;`
`  case Deposit:   deposit(apdu);return;`
`  case Validate:    validate(apdu);return`
`  default:   ISOException.throwIt`
`    (ISO.SW_INS_NOT_SUPPORTED);`
`  }`
 	
`}   // end of process method`
	The main function of the `process` method is to perform an action as specified in the APDU and to return an appropriate response to the terminal. INS byte specifies the type of action needed to be performed.

`private void deposit(APDU apdu) {`
 	
`// access authentication`
`if ( ! pin.isValidated() )`
`  ISOException.throwIt (ISO.SW_PIN_REQUIRED);`
 	
`// Lc byte denotes the number of bytes in the`
`// data field of the comamnd APDU`
`byte numBytes = (byte) (buffer[ISO.OFFSET_LC]);`
 	
`// indicate that this APDU has incoming data and`
`// receive data starting from the offset`
`// ISO.OFFSET_CDATA`
`byte byteRead =`
`(byte)(apdu.setIncomingAndReceive());`
 	
`// it is an error if the number of data bytes`
`// read does not match the number in Lc byte`
`if (byteRead != 1)`
`  ISOException.throwIt(ISO.SW_WRONG_LENGTh);`
 	
`// increase the balance by the amount specified`
`// in the data field of the command APDU.`
`balance = (byte)`
`  (balance + buffer[ISO.OFFSET_CDATA]);`
 	
`// return successfully`
`return;`
 	
`} // end of deposit method`
	The parameter APDU object contains a data field, which specifies the amount to be added onto the balance.
	
	Upon receiving the APDU object from the JCRE, the first 5 bytes (CLA, INS, P1, P2, Lc/Le) are available in the APDU buffer. Their offsets in the APDU buffer are specified in the class ISO. Because the data field is optional, the applet needs to explicitly inform the JCRE to retrieve additional data bytes.
	
	The communication between card and CAD is exchanged between the command APDU and response APDU pair. In the deposit case, the response APDU contains no data field. The JCRE would take the status word 0x9000 (normal processing) to form the correct response APDU. Applet developers do not need to be concerned with the details of constructing the proper response APDU.
	
	When the JCRE catches an Exception, which signals an error during processing the command, the JCRE would use the status word contained in the Exception to construct the response APDU.

<table>
<tr><td><pre>
private void debit(APDU apdu) {

// access authentication
if (! pin.isValidated())
  ISOException.throwIt(ISO.SW_PIN_REQUIRED);

byte numBytes = (byte)(buffer[ISO.OFFSET_LC]);
byte byteRead =
(byte)(apdu.setIncomingAndReceive());

if (byteRead != 1)
  ISOException.throwIt(ISO.SW_WRONG_LENGTH);

// balance can not be negative
if ((balance - buffer[ISO.OFFSET_CDATA]) < 0)
  ISOException.throwIt(SW_NEGATIVE_BALANCE);

balance = (byte)
  (balance - buffer[ISO.OFFSET_CDATA]);

} // end of debit method
</pre></td><td>In the debit method, the APDU object contains a data field, which specifies the amount to be debited from the balance.</td></tr>
<tr><td><pre>
private void getBalance(APDU apdu) {

// access authentication
if (! pin.isValidated())
  ISOException.throwIt(ISO.SW_PIN_REQUIRED);

// inform system that the applet has finished
// processing the command and the system should
// now prepare to construct a response APDU
// which contains data field
apdu.setOutgoing();

//indicate the number of bytes in the data field
apdu.setOutgoingLength((byte)1);

// move the data into the APDU buffer starting
// at offset 0
buffer[0] = balance;

// send 1 byte of data at offset 0 in the APDU
// buffer
apdu.sendBytes((short)0, (short)1);

} // end of getBalance method
</pre></td><td>getBalance returns the Wallet’s balance in the data field of the response APDU.

Because the data field in the response APDU is optional, the applet needs to explicitly inform the JCRE of the additional data. The JCRE uses the data array in the APDU object buffer and the proper status word to construct a complete response APDU.</td></tr>
<tr><td><pre>
private void validate(APDU apdu) {
   // retrieve the PIN data which requires to be
   // valid ated. The user interface data is
   // stored in the data field of the APDU
   byte byteRead =
   (byte)(apdu.setIncomingAndReceive());

   // validate user interface and set the
   // validation flag in the user interface
   // object to be true if the validation.
   // succeeds. If user interface validation
   // fails, PinException would be thrown from
   // the pin.check() method.
   pin.check(buffer, ISO.OFFSET_CDATA, byteRead);

   } // end of validate method
} // end of class Wallet
</pre></td><td>PIN is a method commonly used in smart cards to protect data from unauthorized access.

A PIN records the number of unsuccessful tries since the last correct PIN verification. The card would be blocked if the number of unsuccessful tries exceeds the maximum number of allowed tries defined in the PIN.

After the applet is successfully selected, the PIN needs to be validated first, before any other instruction can be performed on the applet.</td></tr>
</table>

Java Card Applet Installation

Applet installation occurs at the factory or at the office of the issuer and may also occur post-issuance, through a *secure installation* process (if one is defined by the card manufacturer). This process involves downloading a digitally-signed applet, which the JCRE verifies as legitimate, before installing the applet. Applets that are installed through secure downloads cannot contain native method calls since they are not trusted.

Note – Applets with native method calls must be installed at the factory or another trusted environment. This is done for security reasons, since native calls bypass the Java technology security framework and so must be highly trusted before being allowed on the card.

Once installed, Java Card platform classes do not interact directly with the CAD or off-card applications. Installed classes may interact directly with only the JCRE or with other installed classes. The JCRE selects an applet and then passes APDUs to the selected applets. In essence, the JCRE shields the developer from the smart card CPU, the CAD, and the particular ISO communication protocol employed. The JCRE also translates uncaught exceptions thrown by classes or normal return statements in applet methods into standard ISO return values.

The storage for an installed applet cannot be reclaimed; if a newer version of the applet is installed, it occupies a new storage location and the earlier version of the applet becomes unreachable. The Java Card applet can also be made unreachable by removing its reference from the JCRE applet registry table. Once the reference is removed, the applet can no longer be reached.

Installing the Java Card applet causes its static members to be initialized. Java Card technology supports constant static initializers—the initializer cannot execute Java software code, nor can it set the static member to a non-constant (variable) value. Installation also results in a call to the applet's `install()` method (unlike Java applets).

The Applet Class

The Java Card applet extends the `Applet` class. This abstract class has a few methods that must be overridden by the specific implementation of the applet. This section discusses these methods.

When the applet is installed on the smart card, the `install()` method is called once by the JCRE, and never again. The `install()` method must be declared `static` because it is called before the applet is instantiated—the JCRE will **not** call the applet's constructor directly.

The first thing `install()` should do is construct the applet by calling its constructor. This has the effect of allocating and initializing any non-static class instance variables that the applet has declared, and instantiating its non-static methods. Static class variables are allocated and initialized before the constructor is called, when the applet is loaded onto the card.

Since it is up to the applet itself to construct its instance data and methods, the constructor may be declared public, private, or protected. Conceivably, an applet with a public constructor could be constructed by another applet installed on the smart card, but no applet methods or data are visible to non-installed classes, regardless of their permissions.

The APDU object, which is passed to the `install()` method by the JCRE, may contain one-time initialization or personalization data for the applet.

The purse example above inherits methods from the `Applet` class, relying upon their default implementation. The `Applet` class comprises part of the JCRE—a listing for it is provided below to show the methods and their default implementation.

```
public abstract class Applet {
   protected Applet( ) {}
   public static void install (APDU apdu) throws ISOException {
       ISOException.throwIt(ISO.SW_FUNC_NOT_SUPPORTED);
   }
   public void process (APDU apdu) throws ISOException {
       byte buffer[] = apdu.getBuffer();
       ISOException.throwIt(Util.makeShort((byte)0x9F, buffer[1]));
   }
   protected boolean select() {
       return true;
   }
   public void deselect () {}
   protected final void register () {
       AppTable.register (this);
   }
}
```

The Applet class includes a method called `process()` which is the sole mechanism for receiving data from the CAD, once the applet is installed. Information is received by way of an APDU parameter to `process()`. The default implementation of `process()` shown above is merely exemplary. The Java Card platform does not require this behavior of the default implementation of `process()`. Our default implementation of `process()`, in the above example, returns an error code 0x9F along with the instruction byte from the APDU header, indicating that the instruction is not supported.

The `install()` method throws an ISOException (using an exception implementation that maximizes object reuse) by default to indicate that the applet could not be installed. Unless you override `install()`, your applet will not be available on the card.

Remember that it is during the processing of `install()` that all one-time initializations should be performed, including instantiating the applet, registering the applet with the JCRE, and making calls to `new`.

Registering the Applet

In the purse example above, the APDU is passed along to the applet constructor, although there is no requirement for this. The applet constructor does one thing—it calls `register()` to add a reference to the installed applet table within the JCRE.

Every applet must call `register()` during the processing of the call to `install()` to ensure future communication with the JCRE (the JCRE always searches the list of registered applets when deciding where to route messages). Applets depend on the JCRE for all communication and to transfer control to one of the applet's methods when the applet is selected, deselected, or the target of an APDU.

Note – The JCRE is single-threaded, which means that **an infinite loop in any applet hangs all applets** until power to the card is disconnected. When power is returned, the default applet is selected.

Applet Selection

The `Applet` class also includes `select()` and `deselect()` methods. Unlike `process()` and `install()`, these methods do not receive APDUs from the JCRE. The JCRE filters the APDU stream between the installed applets and the CAD, as pictured in FIGURE 3-2, and then calls these methods only when an APDU that selects an installed applet is received.

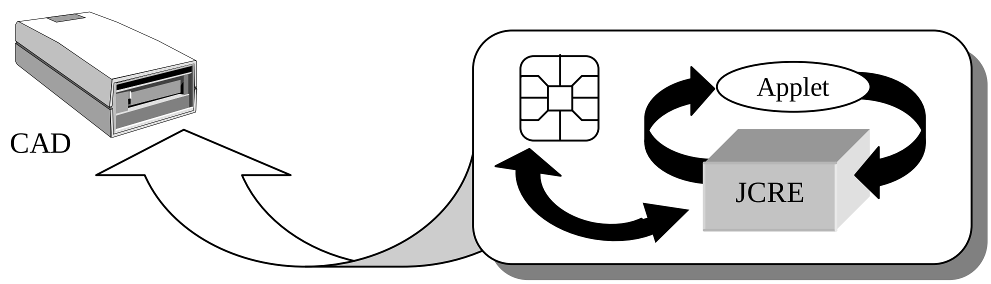

FIGURE 3-2 Applet Selection

As mentioned in the *Overview* chapter, each Java Card applet is assigned a unique Application ID (AID). The AID is used to identify a particular applet for selection. If the JCRE detects a SELECT APDU, it deselects the currently selected applet and then selects the applet indicated by the SELECT APDU data bytes. The JCRE calls the applet's `select()` method just before sending the SELECT APDU to the newly-selected applet's `process()` method. The SELECT APDU causes the JCRE to call the applet twice; one call to `select()`, another call to `process()` to pass the applet the SELECT APDU. It calls the `deselect()` method of the currently selected applet to let the applet know that it is no longer currently selected.

There are two types of SELECT APDUs:

1. For selecting an applet

2. For selecting a file controlled by the selected applet

The two SELECT APDUs are very similar, but are distinguished by the JCRE. Upon receiving a SELECT APDU, the JCRE treats the APDU data as an AID and searches through its applet table. If a match is found, a new applet is selected. Otherwise, the JCRE regards the SELECT APDU as a file-selection APDU and passes it to the currently selected applet's process method.

The default implementation of `select()` returns TRUE, indicating that the applet has been successfully selected to receive subsequent APDUs. Any other return value causes the applet selection to fail, and the JCRE does not forward subsequent APDUs to the applet.

The default implementation of `deselect()` does nothing. Notice that both `select()` and `deselect()` are instance methods. An instance of the `Applet` class must be created in order for the JCRE to call these methods. Since the JCRE does not call the applet constructor, the applet must do so explicitly during the processing of `install()`.

The implementation of `register()` is final and cannot be overridden. The `register()` method inserts an object reference for the applet into the applet registration table within the JCRE. It is through this reference that the JCRE is able to call the non-static `select()` and `deselect()` applet methods.

The following pseudo code summarizes the procedure followed by the JCRE for handling the SELECT APDU:

```
if (SELECT applet APDU)
      call deselect() of currently selected applet
      call select() of applet identified by APDU
      call process() of applet identified by APDU
```

An APDU command is parsed by first calling `APDU.getBuffer()` to retrieve the data packet attached to the APDU. The data packet includes the APDU header and the command data. The JCRE provides constants for quickly parsing the APDU packet. The following code fragment demonstrates the use of the ISO.OFFSET_XXX constants (the interpretation of the header bytes is explained in the *Working with APDUs* section below):

```
byte buf[] = apdu.getBuffer();
byte cla = buf[ISO.OFFSET_CLA];
byte ins = buf[ISO.OFFSET_INS];
byte lc = buf[ISO.OFFSET_LC];
byte p1 = buf[ISO.OFFSET_P1];
byte p2 = buf[ISO.OFFSET_P2];
//get APDU data
Util.arrayCopy(buf, ISO.OFFSET_CDATA, databuf, 0, lc);
```

Note – For proper APDU packet verification, it is a good idea to check all header values, including Lc for valid values.

When the SELECT APDU is received, the applet can retrieve its own AID by calling `System.getAID()`. The AID in the APDU data buffer can be compared against the AID returned from `getAID()` using the `Util.arrayCompare()` method. The method `getAID()` returns an AID object and a byte array, representing the value of the AID and may be extracted by calling `AID.copyTo()`.

Working with APDUs

APDU communications is based on a command-response model, as outlined in *ISO 7816-4*. Generally, an applet sits idle until a command APDU is passed to the applet, requesting some form of processing. The applet processes the command and then returns a response.

The APDU buffer is formatted differently for command APDUs and response APDUs. The general format of a **command APDU** is:

CLA	INS	P1	P2	Lc	Data	Le

The header describes the command for the applet to carry out. The first four bytes of the command APDU represent the header, as follows:

CLA	INS	P1	P2

Where:

- CLA—indicates the class, which identifies if the command is an ISO-conforming message
- INS—indicates the instruction
- P1, P2—indicates additional parameters

The precise values of CLA and INS for different commands are set forth in *ISO 7816-4*.

The other fields of the command APDU are optional (that is, they are not present in some commands). These additional fields define any data passed with the command, as well as the expected length of the response APDU. The other APDU fields are:

Lc	Data	Le

Where:

- Lc—indicates the length of the command data
- Data—indicates the command data
- Le—indicates the length of the expected response

Four different formats of command APDUs are possible, depending upon whether data is included with the command and whether response data is required. When command data is not present and no response is required, the format of the APDU is:

CLA	INS	P1	P2

When command data is present but no response is required, the format of the APDU is:

CLA	INS	P1	P2	Lc	Data

When command data is **not** present, but response data is required, the format of the APDU is:

CLA	INS	P1	P2	Le

When command data is present and response data is required, the format of the APDU is:

CLA	INS	P1	P2	Lc	Data	Le

Note – An Le value of 0 is not the same as an APDU with no Le. An Le value of 0 indicates that the applet should provide all available response data in the response. This would typically occur when the length of the response varies (such as retrieving a variable-length record from a file).

The fifth byte of the APDU (the first byte after the header) may:

- Be missing (indicating that no command or response data is present)
- Contain an Lc value (indicating that command data is present)
- Contain an Le value

The only interpretation of the fifth byte is implicit, by reading CLA and INS and knowing the format of the command they describe.

Applets should never read the Le value directly from the APDU header. Instead, call the APDU method `setOutgoing()` to retrieve the value of Le. Applets should not read from the APDU buffer before calling `setIncomingAndReceive()` or `receiveBytes()` to transfer the incoming data into the APDU buffer.

Responses are always required, even if they contain no data. The format of a **response APDU** is:

Data	SW1	SW2

- Data—response data
- SW1—first status byte
- SW2—second status byte

The first and second status bytes SW1 SW2 are always present in the response. However, the card applet does not set these values directly in the response APDU buffer. Instead, these values are set by the JCRE upon a normal return (SW1=90, SW2=00) or upon an exception thrown by the applet or JCRE.

Since the status bytes are set by the JCRE (upon applet return) and sent with the response data, the applet may not send the response data (or at least, the last block of it) until the applet returns from the `process()` method. This has important implications for buffer management as discussed in the *Receiving APDU Data* section below.

APDU Communication Sequence

A general APDU communication sequence for smart cards is as follows:

1. Receive an APDU object as a parameter to the `process()` method.
2. Parse the APDU command header.
3. Read any APDU data.

> **Note –** The command data will not be in the APDU buffer until it is read by the applet using `setIncomingAndReceive()` or `receiveBytes()`.

4. Process the APDU command and generate any response data.
5. Send any response data from the applet.
6. Perform any additional processing, making sure to not change the response APDU buffer.
7. Return from the `process()` method normally, or throw an ISOException object (the JCRE appends the appropriate status bytes to the response APDU).

Receiving APDU Data

In order to receive command APDU data, the applet must first place the JCRE into receive data mode. Receive mode tells the JCRE to look for data from the CAD and to buffer that data into the APDU data buffer. Data may be received using either a character-oriented protocol or a block-oriented protocol.

As mentioned earlier, the JCRE shields the applet (and the applet developer) from details of the specific protocol implementation. A single method in the `APDU` class, called `setIncomingAndReceive()`, is used to both set the JCRE to receive mode and to receive any available data into the APDU buffer.

The APDU buffer has a minimum size of 37 bytes—an increase in the buffer size is determined by the JCRE developer and is dependent on the memory capacity of the card.

The data is placed in the buffer beginning at offset ISO.OFFSET_CDDATA. The number of bytes read, as pictured in FIGURE 3-3, is determined by the following rule:

If the bytes remaining to be read are:

1. Less than or equal to the buffer length, then read all the remaining bytes

2. Greater than the buffer length, then read up to the buffer length

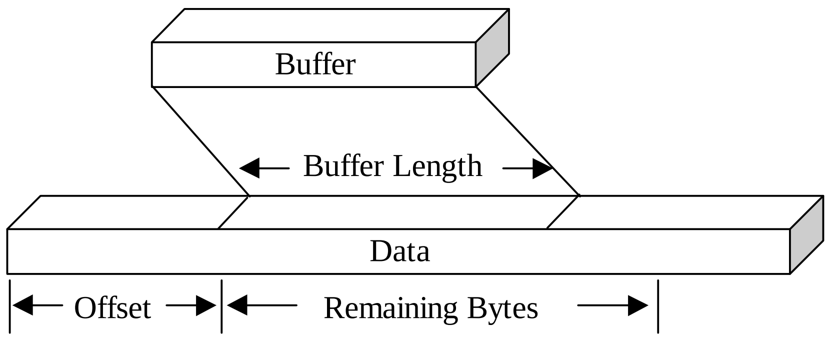

FIGURE 3-3 Buffer Length

To access the data in the APDU buffer, the applet must retrieve a reference to the APDU data buffer by calling the APDU method `getBuffer()`. The data buffer is a byte array whose length can be determined using `buffer.length`. The APDU object received by the applet `process()` method is owned by the JCRE (a single APDU object is shared among all applets on the card).

When there is more data available than can fit in the APDU buffer, the call to `setIncomingAndReceive()` must be followed by one or more calls to the APDU method `receiveBytes()`. This method always reads the available data into the APDU buffer (you cannot specify another applet-supplied buffer). Like `setIncomingAndReceive()`, `receiveBytes()` is guaranteed to return synchronously, with as many bytes as the APDU buffer can hold, only if the remaining bytes fit in the APDU buffer. Otherwise, the method reads as many remaining bytes as will fit in the buffer, and possibly less. The applet should call `receiveBytes()` repeatedly, processing or moving the bytes in the APDU data buffer with each call, until all available data is read. The amount of available data may be determined by checking the Lc byte at `buffer[ISO.OFFSET_LC]`.

For example,

```
byte[] buffer = apdu.getBuffer();
short bytes_left = (short) buffer[ISO.OFFSET_LC];
short readCount = apdu.setIncomingAndReceive();
while (bytes_left > 0) {
      //{process received data in buffer}
      ...
      bytes_left -= readCount;
      //get more data
      readCount = apdu.receiveBytes (ISO.OFFSET_CDDATA);
}
```

Notice that `receiveBytes()` specifies the buffer offset at which to place the data. This enables "buffer splitting" to conserve memory resources. The applet can begin to process received data and build a response APDU, without using separate buffers.

A split APDU buffer used for both reading data and sending a response is shown in FIGURE 3-4.

Response data goes here	Read data here—receiveBytes(buffer.length/2)

FIGURE 3-4 Split APDU Buffer

Note – When the underlying protocol used by the JCRE to communicate with the CAD is block-oriented, the offset used for reading into the APDU buffer (buffer.length/2) must leave enough space for at least one block within the buffer. If the developer does not do this, `receiveBytes()` throws an `APDUException.BUFFER_BOUNDS`.

The block size of the underlying protocol may be determined using the APDU method `getInBlockSize()`. Character-oriented protocols return a block size of 1.

The rules for the behavior of `receiveBytes()` reading may be summarized as follows:

```
if (buffer.length-offset > bytes_left)
      read bytes_left;
if (buffer.length-offset < bytes_left) {
      if (buffer.length—offset < BLOCK_SIZE) throw
            APDUException.BUFFER_BOUNDS;
      else read up to buffer_size;
}
```

The `setIncomingAndReceive()` method must be called before calling `receiveBytes()`. You cannot call `setIncomingAndReceive()` if the JCRE is already in receive mode from a previous call to `setIncomingAndReceive()`—otherwise, an APDUException occurs for APDUException.ILLEGAL_USE.

APDU Responses

Response APDUs may or may not contain data. If the response does not contain data, the applet need not do anything but simply return (or throw an exception). The JCRE generates the appropriate status bytes and sends them to the CAD. If the response requires that the applet return data, then the applet must first place the JCRE into data-send mode and then send the data. Whether or not response data is required, depends upon the particular APDU class and instruction bytes.

When sending data from the applet, you must specify the total number of bytes to send. This total number should always be less than or equal to the expected length of response specified by the Le byte. Once all the response data bytes are sent, the applet must not modify the send buffer for the remainder of the `process()` method. The bytes may not actually be sent until `process()` returns to the JCRE. This enables the JCRE to efficiently combine the response data with the status bytes by holding off on the last send until it is time to send the status bytes.

Note – Data is always sent with status bytes, but in a response APDU, data is optional.

To set the JCRE mode to send, call the APDU method `setOutgoing()`. Unlike the corresponding `setIncomingAndReceive()` method for reading, `setOutgoing()` does not send any bytes; it just sets the mode. Unlike reading, you don't have to send data right away after setting the transfer mode. The applet cannot continue to receive bytes once `setOutgoing()` is called, because the transfer mode is no longer "receive."

The `setOutgoing()` method returns the number of response bytes expected by the application for the command APDU to which the applet is responding. The applet should not respond with more than this number of bytes. The applet must inform the JCRE of the number of response bytes it will be sending, using the APDU method `setOutgoingLength()`.

The `APDU` class contains the `setOutgoingAndSend()` method for:

- Setting the transfer mode to send
- Setting the response data length
- Sending the response bytes

This method closely resembles the operation of `setIncomingAndReceive()`, except that it **sends** bytes, not receives them. The `setOutgoingAndSend()` method uses the APDU buffer for sending bytes. It is also possible to use an applet-supplied and owned buffer to send bytes, which can be bigger than the relatively small APDU buffer.

Response data sent using this method must be short enough to fit in the APDU buffer. The bytes cannot actually be sent until the applet returns from the `process()` method, at which time they are combined with the status bytes—so once this method is called, the applet cannot alter the APDU buffer until `process()` returns.

If you don't want to send data right away after calling `setOutgoing()`, call `sendBytes()` or `sendBytesLong()`. The first method uses the APDU buffer, while the second method uses an applet-supplied buffer. For both methods, you must not alter the contents of the buffer until `process()` returns. If the response data is being prepared in an applet-supplied buffer, use the `apdu.sendBytesLong()` method and provide the applet-supplied buffer as the parameter, to avoid having to copy the applet-supplied buffer to the APDU buffer.

You must call `setOutgoing()` before calling `setOutgoingLength()`, `sendBytes()`, or `sendBytesLong()`. You cannot call `setOutgoingLength()`, `sendBytes()`, or `sendBytesLong()` after calling `setOutgoingAndSend()`. For example, if you have three bytes that need to be sent, use either:

```
byte[] apduBuffer = apdu.getBuffer();

apduBuffer[0] = byte1;
apduBuffer[1] = byte2;
apduBuffer[2] = byte3;

apdu.setOutgoingAndSend(0, 3); //0-offset, 3-number of bytes to send
```

Or:

```
short le = apdu.setOutgoing();

apdu.setOutgoingLength( (short)3 );
apduBuffer[0] = byte1;
apduBuffer[1] = byte2;
apduBuffer[2] = byte3;

apdu.sendBytes ( (short)0 , (short)3 );
```

Return Values

An applet has several options for returning information from the applet `process()` method. The most straightforward method is to simply let the method return. When errors are encountered during APDU processing, use `ISOException.throwIt(short sw)` to return SW errors. ISO SW error constants can be found in the `ISO` class.

Atomicity

With smart cards, there is a risk of losing power at any time during applet execution. A user of the smart card may remove ("tear") the card from the CAD, cutting off power to the card CPU and terminating execution of any applets. The risk of tearing presents a challenge for preserving the integrity of operations on sensitive card data.

Java Card technology supports the notion of "transactions" with commit and rollback capability to guarantee that complex operations can be accomplished *atomically*—either they successfully complete or their partial results are not put into effect. Using transactions, you can guarantee that multiple fields of an object are updated as a unit.

To create a transaction, enclose one or more operations between calls to `System.beginTransaction()` and `System.commitTransaction()`. The pre-transaction values of the destination are written to a temporary location and the updates are made permanent when `System.commitTransaction()` is called. If power is lost during the transaction, a rollback facility is invoked, when power is restored, to return the destination values to their pre-transaction state.

Transactions can be expressly aborted using a call to `System.abortTransaction()`. A transaction must be in progress when `abortTransaction()` is called; otherwise, a `TransactionException` is thrown. Transactions cannot be nested—a call to `beginTransaction` from within a transaction block also results in a `TransactionException`. The values 0 or 1 are returned by `System.getTransactionDepth()`.

Commit Buffer

To support the rollback of uncommitted atomic transactions, the JCRE contains a *commit buffer* where the original contents of the updated locations are stored until the transaction is committed. This buffer also retains the rollback information, should a power loss occur during the commit phase of the transaction. The more operations inside a transaction block, the larger the commit buffer needs to be to accommodate it.

Before attempting a transaction, an applet may check the size of the available commit buffer against the size of the data that requires an atomic update. If sufficient commit capacity is not available, the operations can either be performed (risking power loss during the operations) or postponed until sufficient capacity is available.

The capacity checking options are as follows:

- `System.getMaxCommitCapacity`—returns the total size of the commit buffer
- `System.getUnusedCommitCapacity`—returns the available commit capacity

The `Util` class in `javacard.framework` contains useful methods for copying and filling arrays (`arrayFill`, `arrayFillNonAtomic`, `copyArray`, `copyArrayNonAtomic`). The non-atomic versions of these methods do not participate in transactions, meaning that they do not benefit from rollback and commit capabilities, even when located in a transaction block. To achieve atomicity for array copy and fill, use `arrayCopy` and `arrayFill` inside or outside of a transaction block.

The number of program statements located within a transaction block should be kept to a minimum to avoid exceeding the fixed commit buffer space provided by the implementation.

4. Optimizing Java Card Applets

A major factor influencing the design and features of Java Card applets is the limited availability of program and data memory in the smart card environment. This chapter focuses on the issues related to creating Java Card applets for devices with resource constraints.

The Java Card platform accommodates environments in which only 512 bytes of RAM are available. The JCRE (including the Java Card VM and the system heap) must be contained within the available ROM and the Java Card applets and class libraries need to be stored within the available EEPROM space on the device.

To optimize memory usage, the following restrictions apply when creating Java Card applets:

- A maximum of 127 instance methods in any class (including inherited methods)
- A maximum of 255 bytes of instance data
- Object space is allocated from EEPROM

Note – The Java Card 2.0 Reference Implementation (JC2RI) release contains a simulation environment which allows you to execute Java Card applets in a desktop workstation environment. This environment has memory restrictions which are different from execution in an actual smart card memory environment.

Reusing Objects

Applets must not instantiate objects using `new` with the expectation that their storage will be reclaimed, because the JCRE may not include a garbage collector. The general rule is that in Java Card technology, a single instantiation of an object should be “recycled” repeatedly, with each new use “customizing” the member variables of the object instance. This is a different model than Java technology developers may be accustomed to.

In Java technology, an instance of an object is created as needed, its instance variables are set, and then the object is discarded (typically by going out of scope). In Java Card technology, you should never allow objects allocated with `new` to go out of scope; they will become unreachable, but the storage space they occupy will never be reclaimed.

In Java Card technology, objects should remain in scope for the life of the applet and should be reused by writing new values to their member variables. Note that this does not require all objects to be declared as static.

The following code sample shows how the JCRE implements exceptions in a manner that maximizes object recycling by using the `systemInstance` each time the ISOException `throwIt()` method is used:

```
public class ISOException extends runtimeException {
     private static ISOException systemInstance;
     public ISOException (short sw) { this.reason = sw;}
     public static void throwIt(short sw) {
        if (systemInstance == null) systemInstance = new ISOException(sw);}
     systemInstance.setReason(sw);
     throw systemInstance;
}
```

The exception is thrown by calling the `throwIt()` method with a reason (`sw`) for the exception. The `throwIt()` method is declared static so that there is no need to instantiate an `ISOException` object in order to throw the exception. Instead, simply call `throwIt()`and customize each call with a reason code. The `throwIt()` method in turn invokes `throw` on the exception object. For example:

```
if (buffer[ISO.OFFSET_P1] != 0)
ISOException.throwIt(ISO.SW_INCORRECT_P1P2);
```

Allocating Memory

Memory for primitive types and arrays should be allocated at object-declaration time. Memory for class-member variables is allocated from the system heap, and cannot be reclaimed (unless the smart card implements a garbage collector). Any memory allocated by `new` is taken from the heap. Memory for method variables, locals, and parameters is allocated from the stack and is reclaimed when the method returns.

Typically, `new` should not be called from any applet method other than the `install()` method, unless `new` will only be called sparingly after the applet is installed. The `install()` method is called only once, when the applet is installed on the card, so that a `new` in `install()` results in only a single instance of the object for the lifetime of the applet. It is also recommended that each call to `new` be enclosed in a transaction block (see the *Atomicity* section of the *Creating a Java Card Applet* chapter). This ensures that all of the available memory required by the object is allocated, even if power is lost before `new` returns. Multiple calls to `new` may be enclosed in a single transaction block, but are subject to the limits of available commit capacity (more on this below).

Accessing Array Elements

When accessing an array element, bytecodes are generated to fulfill the array-access instruction. To optimize memory usage, if the same element of an array is accessed multiple times from different locations in the same method, save the array value to a variable on the first access and then access the variable in subsequent accesses. Using the array value as a variable in this way creates more compact bytecodes than re-accessing the array.

5. Files

Support for ISO 7816-4 files is provided by the `javacardx.framework` package. This is an extension package to the core `javacard.*` classes which may not be implemented on all Java Card platforms. Files are identified collections of related data. Files are useful for organizing data into logical groups for storage on the card. The two types of files supported in the javacardx classes, following the definitions of the ISO 7816-4 specification, are:

- Elementary files
- Dedicated files

Elementary and Dedicated Files

Elementary files (EFs) store references to the byte arrays that contain the data and dedicated files (DFs) act as containers for elementary files. The storing of references is done to conserve memory in situations where the applet allocates a byte array and then passes the byte array to the file system to append to a file. The file stores a reference to the byte array and does not allocate new storage for the record. As a result, if the applet subsequently modifies the contents of the byte array, it is in effect modifying the record.

Elementary files can be organized in two ways, as:

- Transparent files—an unstructured sequence of bytes
- Record files— one or more byte-arrays

There is no support in Java Card technology for structured records within files. That is, records are read and written as byte arrays with no regard to the internal field structure of each record. It is left to the applet or external application to interpret the field structure of the byte arrays stored in record files.

Record Files

Record files can be *linear* or *cyclic*.

Linear Files

Linear files contain records organized in an ordered sequence. The records in the file are kept in the order they were inserted, that is, the first inserted record is record number one. The capacity of the linear file may be expanded to include more records (up to the limits imposed by the JCRE).

The records in a linear file may be fixed or variable in length, as pictured in FIGURE 5-1.

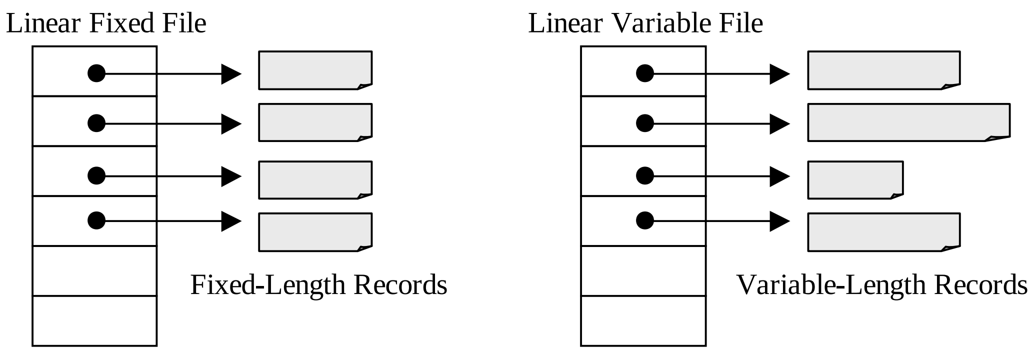

FIGURE 5-1 Linear Files

All of the bytes in a fixed-length record do not need to contain meaningful data, but each record must still contain a fixed number of bytes.

Cyclic Files

In cyclic files, records are organized as a ring (cyclic structure), with fixed and equal record sizes. The number of records in a cyclic file is assigned at file creation time and cannot be changed. Records are in the reverse order as they were inserted into the file—the last inserted record is identified as record number one. Once the file is full, the next append instruction overwrites the oldest record in the file and it becomes the record number one.

The record order of a cyclic file is shown in FIGURE 5-2, before an append command, with RecNum4 being the oldest record:

RecNum4	RecNum3	RecNum2	RecNum1

FIGURE 5-2 Cyclic File Record Order

After the append record command, the new record overwrites the oldest record, RecNum4 and the newly inserted recorded becomes RecNum1, as pictured in FIGURE 5-3.

RecNum1	RecNum4	RecNum3	RecNum2

FIGURE 5-3 Cyclic File after Record Appended

The maximum number of records for any file is 127, and the maximum length of a record is 255 bytes.

The FileSystem Class

The Java Card API includes a `FileSystem` class for processing of file-related APDUs. The `FileSystem` class acts as the master container for a hierarchy of files. The `FileSystem` class directs incoming file-related APDUs to the appropriate member file, then parses and responds to the APDUs. Applets that do not use a `FileSystem` class must manage file-related APDUs on their own.

The following table describes the file types associated with each class:

TABLE 5-1 Classes and File Types

Class	File Type	Description
DedicatedFile	Dedicated	Act as containers for elementary files.
ElementaryFile	Elementary	Store references to the byte arrays that contain the data.
LinearVariableFile	Linear	Stores variable-length linear records.
LinearFixedFile	Linear	Stores fixed-length linear records.
CyclicFile	Cyclic	Stores fixed-length cyclic records.

The **class hierarchy** of the file system is expressed in FIGURE 5-4:

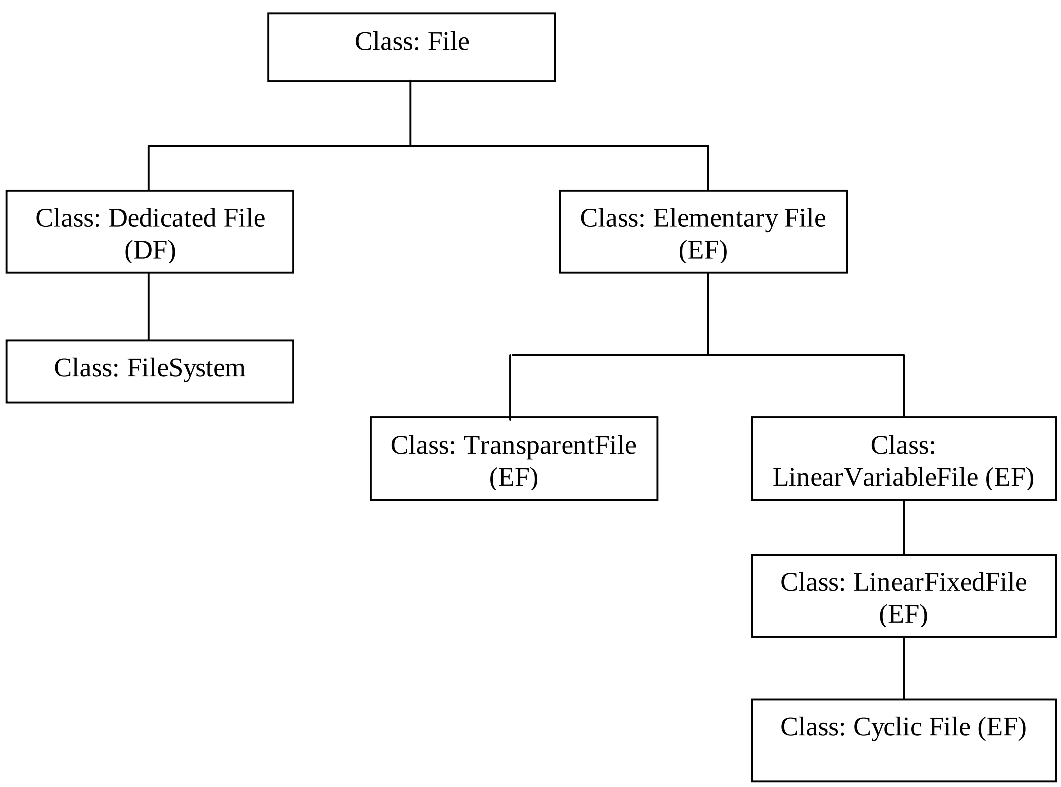

FIGURE 5-4 FileSystem Class Hierarchy

A typical **applet file hierarchy** is shown in FIGURE 5-5, as opposed to the class hierarchy shown above.

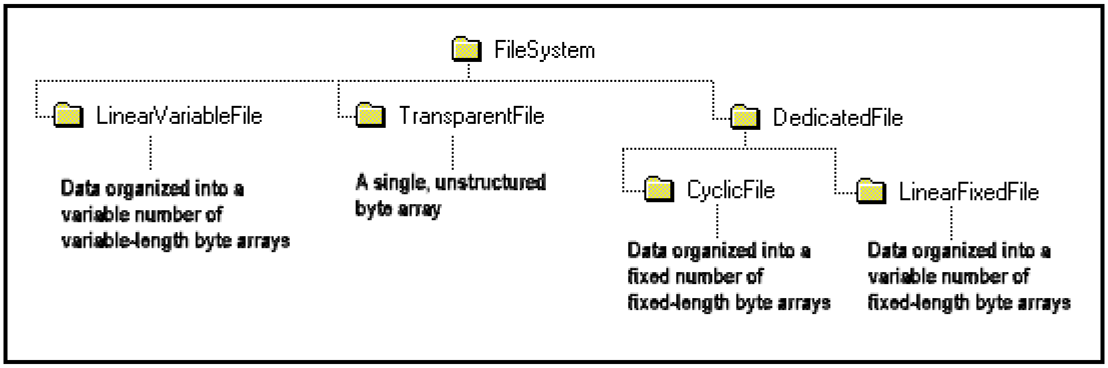

FIGURE 5-5 Applet Data Hierarchy

Notice that the dedicated file contains no data of its own—it only acts as a container for elementary file types that do contain data.

File Operations

The operations that can be performed on files are:

- Creating a new file
- Adding a new file to a dedicated file
- Setting security attributes for a file
- Selecting a file in a file system as the target of implicit operations

A new file is created using the `new` operator on the file's constructor, as follows:

```
cyclic = new CyclicFile (fileID, maxRecords, recordLength);
dedicated = new DedicatedFile (fileID, name, maxChildren);
```

Linear variable files use a constructor with at least the following two parameters:

- A unique file ID
- The maximum number of records the file can hold

The file ID uniquely identifies the file from other files. If the file is part of a group of files contained within a dedicated file, the lower five bits of the file ID (called the Short File Identifier or SFI) are used to locate the file and to distinguish the file from others in the group.

The maximum record count declares the maximum number of records the file can hold. The JCRE uses this value to validate access operations on the file. The JCRE does not use this information to compute or set aside storage for the file. Files in Java Card technology store references to byte arrays, not the byte arrays themselves, but memory for the files is allocated during the installation of the Java Card applet. For example, if a Java Card applet declares a file holding 100 records, each record holding 100 bytes (for a combined total of 10,000 bytes), the VM will prevent the installation of this applet if less than 10,000 bytes of storage are available on the card.

Once the applet is installed, new records may be allocated by the applet (or file system) using the `process()` method during normal card usage (as opposed to during the `install()` method). The maximum record count for linear files may be increased at a later point in time using a call to the `increaseMaxNumRecords()` method. To get the current maximum, call `getMaxNumRecords()`. To get the current number of actual records in the file, call `getNumRecords()`. Fixed-length files (LinearFixedFile and CyclicFile) have an additional parameter to the constructor; the record length. The JCRE uses this information to validate that byte arrays added to the files as records are the correct length.

The constructor for dedicated files (DFs) contains an additional parameter specifying the name of the dedicated file. Unlike elementary files (EFs), dedicated files are identified by a logical name as well as by a short file ID. The logical name of a dedicated file is a byte array, which is not necessarily null-terminated.

Dedicated files (DFs) do not contain records, but they do contain references to other DFs and EFs. These references are called the children of the dedicated file. The constructor for a dedicated file declares the maximum number of children, which may be increased at a later time, but cannot exceed 30.

Adding a new child file to a dedicated file is accomplished by calling the `addChildFile()` method of `DedicatedFile`. For example:

```
dedicated.addChildFile(cyclic);
```

The dedicated file on which the method is invoked becomes the container (parent) for the child file, expressed as a parameter.

File Security

The security attributes of a file are designed to protect the file from external read and write access. Files contain two flags: one to control read permissions and one to control write permissions. Permissions are set independently of one other. The available permissions are shown in the table below.

TABLE 5-2 File Access Permission Flags

Permission	Meaning for Reading	Meaning for Writing
ALLOW_ANY	Reading of file data permitted	Writing of file data permitted
ALLOW_NONE	Reading of file data not permitted	Writing of file data not permitted
ALLOW_AUTH1	Reading only allowed if AUTH1 flag is set in the file system to which this file belongs	Writing only allowed if AUTH1 flag is set in the file system to which this file belongs
ALLOW_AUTH2	Reading only allowed if AUTH2 flag is set in the file system to which this file belongs	Writing only allowed if AUTH2 flag is set in the file system to which this file belongs

The last two flags, ALLOW_AUTH1 and ALLOW_AUTH2, are used when the file belongs to a `FileSystem` class. For example, suppose a file only supports reading and writing after verification of the user's PIN number. The applet verifies the PIN and sets the AUTH1 flag in the `FileSystem` class to TRUE. The applet then would set the read and write permissions of the file to ALLOW_AUTH1.

When reading and writing requests on the file are made through the `FileSystem` class, an AND operation is performed on the two flags. When the file is accessed through the `FileSystem` methods, the applet is responsible for ensuring that the proper conditions are met for granting read and write access to the file.

Only one file belonging to the file system hierarchy may be selected as the current file. The current file is the target of file operations that do not explicitly declare which file they operate upon. Each file system may contain one current elementary file and one current dedicated file. The `setCurrentElementaryFile()` and `setCurrentDedicatedFile()` methods are used to set the current elementary and dedicated files, respectively.

Note – Once files are created, they remain on the card throughout the life of the card.

Finding Files

The file-finding methods are all part of the `DedicatedFile` class because dedicated files acts as containers for all DFs and EFs. Files are located in several ways, but the most common method for locating a file is by using the file's short file-ID, which is set when the file object is constructed. Searching by file ID applies to both dedicated and elementary files because both types have file IDs.

The `findFile()` method takes the file ID as an argument and lets the applet specify the scope of the search using a flag as defined in the table below.

TABLE 5-3 File Flags

Flag	Scope
FIND_CHILD_EF	Search the children of the current dedicated file for a matching elementary file.
FIND_CHILD_DF	Search the children of the current dedicated file for a matching dedicated file.
FIND_CHILD	Search the children of the current dedicated file for a matching elementary or dedicated file.

Up to 30 elementary files may be uniquely identified among the direct children of a dedicated file (the values 0 and 31 are reserved). The last five bits of the file ID, referred to as the Short File Identifier (SFI) are used to search for an elementary file, using the call to `findElementaryFile()`.

Dedicated files, unlike elementary files, are identified by both a file ID and a logical name. To search for a dedicated file that is a child of the current dedicated file, call `findDedicatedFile()` and specify the logical name of the file. Another way to locate a file is by using the file's child id. The child id is just a value identifying the sequential order in which the file was added to the current dedicated file. The `getChildFile()` method locates a file using its child id.

The `FileSystem` class allows an applet to select a file using the file's object reference. The method `selectFile()` takes an object reference as a parameter and makes the referenced file the current file. An applet developer can bypass the `FileSystem` class and access the dedicated file directly by using the file ID (FID). For example, if the file structure is:

```
DedicatedFile master with FID (0x1234);
```

where master contains two files, namely a DF and an EF, then the applet would invoke:

```
df.findFile (FIND_XXX, 0x1234)
```

If FIND flag is FIND_CHILD, then the `findFile()` method returns null. If the FIND flag is FIND_ANY, then the `findFile()` method would search all the siblings, children, and parent of the associated DF, and return the master file with FID = 0x1234.

Record Operations

Files (except for dedicated files) are containers for data. Transparent files store data as a single unstructured stream of bytes. Other files organize data into records. The Java Card API enables the following operations on files using records:

- Add a record to a file.
- Write data to a record in a file.
- Read data from a record.
- Erase the data in a record.

As with files, once a record is created, it remains within the file throughout the life of the card. Erasing a record is not the same as deleting it. Erasing simply resets the bytes in the record to zero.

Records may be added to files of type LinearVariableFile and LinearFixedFile using the `addRecord()` method. Two varieties of `addRecord()` are provided: one in which the applet allocates the byte array for the record to add, another which relies upon the JCRE to allocate the byte array for the record. With either method, the byte array does not become part of the file—only a reference to the array is added to the file.

According to ISO 7816-4, files of type `CyclicFile` do not support the adding of records. Transparent files and dedicated files do not contain records.

To write data to a record, use the following steps:

1. Call the `getRecord()` method to retrieve the byte array of the record.

2. Change the byte array, which in effect writes data to the record.

Finding Records

Once you have located a file, you must then locate the record to read or write. Records are located using either the sequential record number or by searching on the record key. For linear files, the record number is a sequential number valued 1 to N, where N is the number of records in the file. The record number is assigned when the record is added to the file and never changed.

For cyclic files, the record number varies from 1 to N, where N is the number of records in the file. Unlike linear files, the record numbers in a cyclic file are assigned in the reverse order from how the records were written—the last record written to a cyclic file is assigned the record number 1 and the least-recently-written record is assigned record number N. Use the `getRecord()` method and specify the record number to retrieve the byte array for a particular record. See the *Elementary and Dedicated Files* section above for details on the cyclic file.

The record key is a combination of the first and second bytes of the record. Records are located by key, using the `findRecord()` method. The `findRecord()` method operates in both a relative and absolute mode. In relative mode, the search begins at the current record and either moves forward or backward from there. In absolute mode, the search begins at the beginning of the file and moves forward or progresses backward from the end of the file. A direction flag specifies the search mode, as indicated in the table below.

TABLE 5-4 Search Mode Direction Flags

Flag	Mode
DIRECTION_FIRST	Search forward from start of file.
DIRECTION_LAST	Search backward from end of file.
DIRECTION_NEXT	Search forward from current record.
DIRECTION_PREV	Search backward from current record.

Managing Files with the FileSystem Class

The Java Card API contains a special class called `FileSystem` for managing file-related APDUs. The `FileSystem` class extends `DedicatedFile`, acting as a control point for commands referencing the elementary (data) and dedicated (container) files it contains. See *The FileSystem Class* section above for more information on the `FileSystem` class.

The `FileSystem` class provides a default implementation for:

- Reading, writing, updating, and erasing data in transparent files
- Reading, writing, updating, and appending records in record files
- Reading and writing to files with records in Tag-Length-Value format

The `FileSystem` class also provides a security mechanism for providing a measure of access control to individual files. The security mechanism is based upon two flags, AUTH1 and AUTH2. The `setAuthFlag()` method is used to set the value of AUTH1 and AUTH2 to either TRUE or FALSE. The `getAuthFlag()` method is used to get the value of these flags.

Central to the use of the `FileSystem` class is the concept of the current file and current record. An APDU may omit the specification of a target file for its operation—in this situation, the current file becomes the target. Likewise, an APDU may not specify a target record for its operation—in this situation, the current record becomes the target. The current data file for an APDU operation is always relative to the currently selected dedicated file within the file system.

Since the `FileSystem` class is itself a dedicated file, it may contain other dedicated files, which, in turn, may contain elementary files. The current record is always relative to the current elementary file. The `FileSystem` class contains methods for setting and getting the current dedicated file, the current elementary file, and the current record, as outlined in the table below.

TABLE 5-5 FileSystem Methods

Method	Description
getCurrentDedicatedFile()	Gets current dedicated file.
getCurrentElementaryFile()	Gets current elementary file.

getCurrentRecNum()	Gets current record number.
setCurrentDedicatedFile()	Sets current dedicated file.
setCurrentElementaryFile()	Sets current elementary file.
setCurrentRecNum()	Sets current record number.

These set methods are not the only mechanism for setting the current elementary file or record number. Certain ISO commands change the current elementary file or record. The following table shows the `FileSystem` methods used in processing various ISO commands and the resulting outcome (a CLA of 0 is assumed for all ISO commands):

TABLE 5-6 FileSystem Methods

ISO Command	INS	Description	FileSystem Method	Resulting Outcome
READ BINARY	B0	Read data from a transparent file	readBinary()	Makes target current EF and resets the current record number to 0.
WRITE BINARY	D0	Write data to a transparent file	writeBinary()	Makes target current EF and resets the current record number to 0.
UPDATE BINARY	D6	Update data in a transparent file	updateBinary()	Makes target current EF and resets the current record number to 0.
ERASE BINARY	0E	Mark data as erased in a transparent file	eraseBinary()	Makes target current EF.
APPEND RECORD	E2	Append a record to a record-oriented file	appendRecord()	Makes target current EF. Make appended record current record.
READ RECORD	B2	Read data from a record-oriented file	readRecord()	Makes target current EF. Make read record current record.
WRITE RECORD	D2	Write data to a record-oriented file	writeRecord()	Makes target current EF. Make written record current record.
UPDATE RECORD	DC	Update data in a record-oriented file	updateRecord()	Makes target current EF. Make updated record current record.
GET DATA	CA	Retrieve Tag-Length-Value (TLV) formatted data from current file.	getData()	None
PUT DATA	DA	Set Tag-Length-Value (TLV) formatted data into current file.	putData()	None
SELECT	A4	Selects a file as the current EF or DF	select()	If target is an EF, selects the target's DF as the current DF.

These methods are protected methods and are not meant to be invoked by applets. These methods not only process the command APDU, but they also build a response APDU with status bytes and send it out.

Note –An applet should not modify the command APDU buffer in any way once these methods return because the JCRE uses the command buffer for the response. The last response (including status bytes) may not be sent until the applet returns from its `process()` method.

Since the values of the CLA and INS bytes are standard for the APDUs processed by the `FileSystem` class, there is no reason to require the applet to first parse the APDU command header for the CLA and INS bytes, perform a `switch()` statement, and call the appropriate `FileSystem` method. Instead, the `FileSystem` class provides the `process()` method for performing this processing (not to be confused with the applet `process()` method). Unless the applet needs to perform non-standard processing for one of the command APDUs listed in the table above, there is no reason to parse the APDU command buffer or call any of the `FileSystem` methods set forth in the table. Use `FileSystem.process()` to route the APDUs to the appropriate methods for standard processing. For example:

```
//The following code fragment shows how an applet typically delegates
//various APDUs for processing, particularly how an applet passes
//ISO file-oriented APDUs to FileSystem for it to handle.

buffer = apdu.getBuffer();
FileSystem fs = new FileSystem(4);

if ( buffer[ISO.OFFSET_CLA] == 0 ) {  // ISO file-oriented APDU
  switch (buffer[ISO.OFFSET_INS] {
    case XX:
         // what goes here is code for the applet to handle any ISO file-
         // oriented APDU instructions or and APDU
         // instructions that the FileSystem method does not support
         return;
    default:
         // standard file system APDU instruction
         fs.process(apdu);
         return;
  }
}

// APPLET_CLA - applet CLA byte, a byte number specifically assigned to
// this applet
if ( buffer[ISO.OFFSET_CLA] == APPLET_CLA )
// applet-specific APDU instructions are handled here
   return;
}

// Cannot handle other APDU instructions where CLA byte is
// neither APPLET_CLA nor 0.
ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
```

Developers interested in handling the file system APDUs themselves (for example, when the `javacardx.framework` package is not implemented on the platform) should refer to *ISO 7816-4* for a complete reference on the command structure of these APDUs and the expected responses.

6. Cryptography

Cryptography in Java Card technology is provided by two extension packages: `javacardx.crypto` and `javacardx.cryptoEnc`. The `cryptoEnc` package keeps DES encryption capability separate. This allows implementers to exclude DES from the `crypto` package to comply with US export restrictions. Neither package is part of the core required Java Card API packages and so cryptography may not be supported on some Java Card-based smart cards.

Understanding the Java Card API cryptography requires an understanding of basic cryptography concepts that are discussed in this chapter.

Cryptography Concepts

Cryptography supports two basic functions: *privacy* and *authentication*. Privacy means the secure exchange of information between two or more locations without the risk of eavesdropping by parties who are not the intended recipients of the message. Authentication means proving that a message is actually from the expected party and not from an imposter. Authentication also means proving that a message was not altered during transmission.

A fundamental concept of cryptography is the *key*. A key is a string of bits that is fed into an algorithm for encoding or decoding information. Cryptography is based upon the concept that certain algorithms for decoding information will only work if provided with a key identical or related to the key used in the algorithm for encoding the information.

When an identical key is used to encode and decode, the algorithms are said to be *symmetric*. The following example illustrates how two parties (A and B) communicate using symmetric algorithms. Party A creates a message *m* and applies the encoding algorithm to it. The output of the encoding algorithm is the encoded message *c*:

```
c = f(ks, m)
```

The symbol ks represents the key used by both the encoding and decoding algorithms. The encoded message *c* is transmitted to Party B. Party B decodes the message by applying the decoding algorithm to *c* to reproduce the original message *m*:

```
m = g(ks, c)
```

Symmetric algorithms are fast and efficient. However, both the sender and receiver of the information must possess the key to exchange encoded information. For this reason, the key in a symmetric algorithm is sometimes called a *shared key*. This creates a problem—how can the two parties exchange the shared key without running the risk of a third party intercepting the key and gaining access to their subsequent conversations?

To solve this problem, *asymmetric* algorithms were developed. Using asymmetric algorithms, one key is used in the encoding algorithm, and a different key is used in the decoding algorithm. The two keys are not identical, but they are related. Most importantly, the relationship between the keys is such that the key for decoding cannot be derived from the key for encoding. This important property makes it possible to publish the key for encoding to the world, while keeping the key for decoding private. The following example illustrates how two parties (A and B) communicate using asymmetric algorithms. Party A creates a message *m* and applies the encoding algorithm to it. The output of the encoding algorithm is the encoded message *c*:

```
c = f (k1, m)
```

The symbol k1 represents the *public key* used by the encoding algorithm for a particular individual (Party B in this case). The k1 key can only be used to encode messages to B, not decode them. The encoded message *c* is transmitted to Party B. Party B decodes the message by applying the decoding algorithm to *c* to reproduce the original message *m*. The decoding algorithm uses the second key, k2:

```
m = g(k2, c)
```

Only Party B can decode messages encoded with key k1, because only Party B possesses the key k2—which is why the term *private key* is used to describe k2.

Asymmetric algorithms are slower than symmetric algorithms, but they are more secure because there is no need for the communicating parties to exchange private keys. A hybrid approach that uses the best features of both symmetric and asymmetric algorithms first uses the recipient's public key to encode a shared private key. The shared private key is used to encode the bulk of the message and then the encoded, shared private key is attached to the encoded message and sent. The recipient decodes the shared private key first, using the recipient's private key. Next, the recipient uses the decoded, shared key to decode the bulk of the message.

The Java Card API supports both symmetric and asymmetric algorithms.

Symmetric Keys

The `javacardx.crypto` package contains support for creating keys for use in symmetric algorithms. The base class for symmetric keys is `SymKey`, which extends the basic `Key` class used for all keys. Before a symmetric key can be used, its value must be set using the `setKey()` method. This method sets the bit string for the key from a byte array, provided as a parameter. Unlike asymmetric algorithms, it is not necessary to create two distinct, related keys.

The symmetric algorithms supported by the Java Card API are block oriented. That means that the algorithms encode information one block at a time. A typical block size is 8 bytes. The Java Card API supports two modes of block encoding: Electronic Code Book (ECB) and Cipher Block Chaining (CBC). ECB mode takes one block of information at a time, encodes it, and produces an encoded block, as illustrated in FIGURE 6-1.

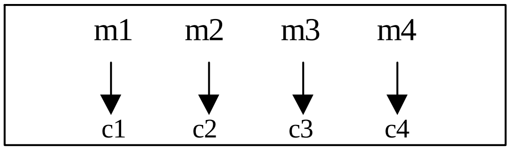

FIGURE 6-1 ECB Diagram

CBC mode takes one block of information at a time, combines it with the previously encoded block, and produces an encoded block from the combination, as illustrated in FIGURE 6-2.

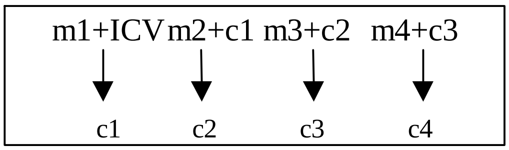

FIGURE 6-2 CBC Diagram

CBC mode has the advantage of hiding block-sized repetitions in the original message *m*. Notice that for the first block, the CBC algorithm does not have a block of encoded information to add with the first block of the message. A value called the initial chaining vector (ICV) is added to the first message block to overcome this problem.

The Java Card API includes four classes for implementing DES symmetric encoding and decoding. The first class is `DES_Key`. This class is used for single DES decoding only. Single DES refers to the single-pass DES algorithm, in which the message is encoded once using a single key. A second class, called `DES3_Key`, implements triple DES decoding only. Triple DES refers to a three-stage application of the DES algorithm in which the message is encoded three times using two single-length keys. Triple DES is considered more secure in some situations than single DES.

Encoding functionality for the two DES algorithms is added by the `DES_EncKey` and `DES3_EncKey` classes, which extend the `DES_Key` and `DES3_Key` decoding classes, respectively. These two classes are part of the `javacardx.cryptoEnc` package, not the `javacardx.crypto` class. The reason for keeping them separate, as mentioned earlier, is that not all Java technology smart cards will have DES encryption capability due to export restrictions on this technology.

Verification of Symmetrically-Encoded Messages

Messages encoded using symmetric algorithms are verified by attaching a *message authorization code* (MAC) to the message. The MAC serves to verify that the message has not been altered from its original form. The MAC is created as a function of the message data and the key, using the `generateMAC()` method. To authenticate a message with an attached MAC, call `verifyMAC()`. Verification of the integrity of a message with an attached MAC is a function of the MAC value, the message data, and the key. The Java Card API supports the creation and verification of MACs using the CBC encoding method.

The CBC algorithm acts in a ripple fashion by employing the block output of the previous iteration of the algorithm to the next block input. The block output of the final iteration of the algorithm is a unique representation of the entire message, incorporating information from all previous blocks in the message. This final output is ideally suited for use as a MAC. If the message is altered in any way during transmission, the attached MAC value will not equal the MAC value generated on the altered text.

Asymmetric Keys

Unlike symmetric keys, asymmetric algorithms use two keys: one public, one private. This key-pair has a mathematical relationship to one another. The base class for all asymmetric keys is `AsymKey`, and like `SymKey`, `AsymKey` is an extension of the generic `Key` class. Creating a symmetric key is easy—simply set the bit string for the key using the `setKey()` method. Creating asymmetric key pairs is more complicated, because the key pair must be computed mathematically from certain key parameters.

To understand how to create asymmetric keys, one must first understand how they are computed. Since Java Card technology implements the RSA asymmetric algorithm, we shall discuss key creation in terms of this algorithm. Generating an RSA key-pair first requires two large prime numbers, *p* and *q*. These two numbers are multiplied to produce a modulus, *n*:

$$n = pq$$

Next, a value *e* is chosen that meets the following mathematical relationships:

$e<n$ and e has no common factors with the values $(p-1)$ or $(q-1)$

The value *e* is called the public exponent. Next, a value `d` is chosen such that the value `ed-1` is roundly divisible by the value `(p-1)(q-1)`. *d* is called the private exponent. The pair (n,e) form the public key, and the values (n,d) form the private key. The values of *d*, *p*, and *q* must be kept strictly secret, while the values of *n* and *e* are made public. To encode a message with the public key, use the following formula:

$$c = m^e \bmod n$$

To decode a message using the private key, use the following formula:

$$m = c^d \bmod n$$

The formulas just described are the basis of one common form of the RSA algorithm, known as the *modulus-exponent* form. Another common form is called the *Chinese Remainder Theorem* (CRT) form. The CRT form also has its roots in the prime factors *p* and *q*, but it is computed differently, using a formula not relevant to the discussion here. From a programming point of view, one needs only to realize that instead of setting the values (n,e) for the public key and (n,d) for the private, one must set the following values for the private key only:

p
q
$p^{-1} \bmod q$
$d \bmod (p-1)$
$d \bmod (q-1)$

Java Card technology implements the RSA algorithm in both its modulus-exponent form and the CRT form. The Java Card API contains classes for both public and private keys used in asymmetric algorithms. All public keys descend from the `PublicKey` class. All private keys descend from the `PrivateKey` class. Specifically, `RSA_PublicKey` defines a public key for use with the RSA algorithm in its modulus-exponent form and `RSA_PrivateKey` defines a corresponding private key.

To create a public key in this form, you must specify the modulus *n* and the exponent *e*. To create a private key, you must specify the modulus *n* and the exponent *d*. The class `RSA_CRT_PrivateKey` implements an RSA private key in the CRT form. To create a private key in this form, you must specify all of the five parameters listed above for the CRT algorithm. A CRT private key can be paired with a public key of the modulus-exponent form.

Authentication and Verification

Asymmetric algorithms go beyond symmetric algorithms in that they allow the recipient of a message to verify not only the integrity of a message, but also its source. Verifying the source of a message is called *authentication*. To create an authentic message, the sender of a message first creates a *hash value* for the message. The hash value (also known as a message digest) is a string of bits that uniquely identifies the message. The sender then encodes the hash value with their private key.

Next, the bulk of the message is encoded using the recipient's public key. The encoded hash value is attached to the encoded message and acts as a *digital signature* for that message. The recipient of the message first decodes the bulk of the message using the recipient's private key. Then, the recipient compares the digital signature with the message, using a simple mathematical relation involving:

- The sender's public key
- The signature
- The message

The `Sha1MessageDigest` class creates a secure hash value for a message suitable for use in digital signatures. This class extends the base `MessageDigest` class which incorporates general hash table functionality. Notice from the above discussion that signatures are only created using the sender's private key—the methods for signing messages in the Java Card API are only included in classes that extend `PrivateKey`. The `sign()` method is part of this class and takes as parameters the sender's private key and the message data to sign.

Verification/authentication requires only the sender's public key—only classes extending the `PublicKey` class include a method for verification. The `verify()` method is part of this class and includes parameters for the hash value of the message to verify, along with the signed data itself.

Glossary

AID is an acronym for Application IDentifier as defined in ISO 7816-5.

APDU is an acronym for Application Protocol Data Unit as defined in ISO 7816-4.

API is an acronym for Application Programming Interface. The API defines calling conventions by which an application program accesses the operating system and other services.

Applet the basic unit of selection, context, functionality, and security in Java Card technology.

Applet developer refers to a person creating an applet using the Java Card 2.0 API.

Applet execution context. The JCRE keeps track of the currently selected applet as well as the currently active applet. The currently active applet value is referred to as the applet execution context. When a virtual method is invoked on an object, the applet execution context is changed to correspond to the applet that owns that object. When that method returns, the previous context is restored. Invocations of static methods have no effect on the applet execution context. The applet execution context and sharing status of an object together determine if access to an object is permissible.

Atomic Operation is an operation that either completes in its entirety (if the operation succeeds) or no part of the operation completes at all (if the operation fails).

Atomicity refers to whether a particular operation is atomic or not and is necessary for proper data recovery in cases where power is lost or the card is unexpectedly removed from the CAD.

CAD is an acronym for Card Acceptance Device. The CAD is the device in which the card is inserted.

Cast is the explicit conversion from one data type to another.

Class is the prototype for an object in an object-oriented language. A class may also be considered a set of objects which share a common structure and behavior. The structure of a class is determined by the class variables which represent the state of an object of that class and the behavior is given by a set of methods associated with the class.

Classes are related in a class hierarchy. One class may be a specialization (a “subclass”) of another (one of its “superclasses”), it may be composed of other classes, or it may use other classes in a client-server relationship.

EEPROM is an acronym for Electrically Erasable, Programmable Read Only Memory.

Framework is the set of classes which implement the API. This includes core and extension packages. Responsibilities include dispatching of APDUs, applet selection, managing atomicity, and installating applets.

Garbage Collection is the process by which dynamically allocated storage is reclaimed during the execution of a program.

Java Card Runtime Environment (JCRE) consists of the Java Card Virtual Machine, the framework, and the associated native methods.

JC2RI is an acronym for the Java Card 2.0 Reference Implementation.

JCRE Developer refers to a person or company creating a vendor-specific framework using the Java Card 2.0 API.

JDK is an acronym for Java Development Kit. The JDK is a Sun Microsystems, Inc. product which provides the environment required for programming in the Java software language. The JDK is available for a variety of platforms, for example Sun Solaris and Microsoft Windows.

Instance Variable, in object-oriented programming, is when a different value may exist for each object of a class.

Instantiation, in object-oriented programming, means to produce a particular object from its class template. This involves allocation of a data structure with the types specified by the template, and initialization of instance variables with either default values or those provided by the class’s constructor function.

MAC is an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.

Method is the name given to a procedure or routine, associated with one or more classes, in object-oriented languages.

Namespace is a set of names in which all names are unique.

Object-Oriented is a category of programming languages and techniques based on the concept of an “object” which is a data structure encapsulated with a set of routines, called “methods,” which operate on the data.

Objects, in object-oriented programming, are unique instances of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

Package is a Java software namespace and can have classes and interfaces. A package is the smallest unit of Java software that can be processed by the JCC utility and installed on a Java technology smart card.

Runtime Environment, see JCRE.

Transaction is an atomic operation where the developer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

