Java Card™ 2.1 Virtual Machine Specification

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300 fax 415 969-9131

Final Revision 1.0, March 3, 1999

Contents

Figures

Tables

Preface

Java Card™ technology combines a subset of the Java programming language with a runtime environment optimized for smart cards and similar 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 devices such as smart cards.

The Java Card platform is defined by three specifications: this *Java Card™ 2.1 Virtual Machine Specification*, the *Java Card™ 2.1 Application Programming Interface*, and the *Java Card™ 2.1 Runtime Environment (JCRE) Specification*.

This specification describes the required behavior of the Java Card 2.1 Virtual Machine (VM) that developers should adhere to when creating an *implementation*. An implementation within the context of this document refers to a licensee's implementation of the Java Card Virtual Machine (VM), Application Programming Interface (API), Converter, or other component, based on the Java Card technology specifications. A Reference Implementation is an implementation produced by Sun Microsystems, Inc. Application software written for the Java Card platform is referred to as a Java Card applet.

Who Should Use This Specification?

This document is for licensees of the Java Card technology to assist them in creating an implementation, developing a specification to extend the Java Card technology specifications, or in creating an extension to the Java Card Runtime Environment (JCRE). This document is also intended for Java Card applet developers who want a more detailed understanding of the Java Card technology specifications.

Before You Read This Specification

Before reading this document, you should be familiar with the Java programming language, the Java Card technology specifications, 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`.

How This Book Is Organized

Chapter 1, "Introduction," provides an overview of the Java Card Virtual Machine architecture.

Chapter 2, "A Subset of the Java Virtual Machine," describes the subset of the Java programming language and Virtual Machine that is supported by the Java Card specification.

Chapter 3, "Structure of the Java Card Virtual Machine," describes the differences between the Java Virtual Machine and the Java Card Virtual Machine.

Chapter 4, "Binary Representation," provides information about how Java Card programs are represented in binary form.

Chapter 5, "The Export File," describes the Converter `export` file used to link code against another package.

Chapter 6, "The CAP File Format," describes the format of the `CAP` file.

Chapter 7, "Instruction Set," describes the byte codes (opcodes) that comprise the Java Card Virtual Machine instruction set.

Chapter 8, "Tables of Instructions," summarizes the Java Card Virtual Machine instructions in two different tables: one sorted by Opcode Value and the other sorted by Mnemonic.

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

Prerequisites

This specification is not intended to stand on its own; rather it relies heavily on existing documentation of the Java platform. In particular, two books are required for the reader to understand the material presented here.

[1] Gosling, James, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996, ISBN 0-201-63451-1 – contains the definitive definition of the Java programming language. The Java Card 2.1 language subset defined here is based on the language specified in this book.

[2] Lindholm, Tim, and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1996, ISBN 0-201-63452-X – defines the standard operation of the Java Virtual Machine. The Java Card virtual machine presented here is based on the definition specified in this book.

Related Documents

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

- *Java Card™ 2.1 Application Programming Interface*, Sun Microsystems, Inc.
- *Java Card™ 2.1 Runtime Environment (JCRE) 2.1 Specification*, Sun Microsystems, Inc.
- *Java Card™ 2.1 Applet Developer's Guide*, 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.
- *The Java™ Class Libraries: An Annotated Reference (Java Series)* by Patrick Chan and Rosanna Lee. Addison-Wesley, ISBN: 0201634589.
- *ISO 7816 International Standard*, First Edition 1987-07-01.
- *EMV '96 Integrated Circuit Card Specification for Payment Systems*, Version 3.0, June 30, 1996.

Ordering Sun Documents

The SunDocs™ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	Java code, Java keywords or variables, or class files.	The `token` item of a `CONSTANT_StaticFieldref_info` structure ...
bytecode	Java language bytecodes	*invokespecial*
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename.*
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Acknowledgements

Java Card technology is based on Java technology. This specification could not exist without all the hard work that went into the development of the Java platform specifications. In particular, this specification is based significantly on the *Java™ Virtual Machine Specification*. In order to maintain consistency with that specification, as well as to make differences easier to notice, we have, where possible, used the words, the style, and even the visual design of that book. Many thanks to Tim Lindholm and Frank Yellin for providing a solid foundation for our work.

CHAPTER 1

Introduction

1.1 Motivation

Java Card technology enables programs written in the Java programming language to be run on smart cards and other small, resource-constrained devices. Developers can build and test programs using standard software development tools and environments, then convert them into a form that can be installed onto a Java Card technology enabled device. Application software for the Java Card platform is called an applet, or more specifically, a Java Card applet or card applet (to distinguish it from browser applets).

While Java Card technology enables programs written in the Java programming language to run on smart cards, such small devices are far too under-powered to support the full functionality of the Java platform. Therefore, the Java Card platform supports only a carefully chosen, customized subset of the features of the Java platform. This subset provides features that are well-suited for writing programs for small devices and preserves the object-oriented capabilities of the Java programming language.

A simple approach to specifying a Java Card virtual machine would be to describe the subset of the features of the Java virtual machine that must be supported to allow for portability of source code across all Java Card technology enabled devices. Combining that subset specification and the information in the *Java Virtual Machine Specification*, smart card manufacturers could construct their own Java Card implementations. While that approach is feasible, it has a serious drawback. The resultant platform would be missing the important feature of binary portability of Java Card applets.

The standards that define the Java platform allow for binary portability of Java programs across all Java platform implementations. This “write once, run anywhere” quality of Java programs is perhaps the most significant feature of the platform. Part

of the motivation for the creation of the Java Card platform was to bring just this kind of binary portability to the smart card industry. In a world with hundreds of millions or perhaps even billions of smart cards with varying processors and configurations, the costs of supporting multiple binary formats for software distribution could be overwhelming.

This *Java Card 2.1 Virtual Machine Specification* is the key to providing binary portability. One way of understanding what this specification does is to compare it to its counterpart in the Java platform. The *Java Virtual Machine Specification* defines a Java virtual machine as an engine that loads Java `class` files and executes them with a particular set of semantics. The `class` file is a central piece of the Java architecture, and it is the standard for the binary compatibility of the Java platform. The *Java Card 2.1 Virtual Machine Specification* also defines a file format that is the standard for binary compatibility for the Java Card platform: the `CAP` file format is the form in which software is loaded onto devices which implement a Java Card virtual machine.

1.2 The Java Card Virtual Machine

The role of the Java Card virtual machine is best understood in the context of the process for production and deployment of Java Card software. There are several components that make up a Java Card system, including the Java Card virtual machine, the Java Card Converter, a terminal installation tool, and an installation program that runs on the device, as shown in Figures 1-1 and 1-2.

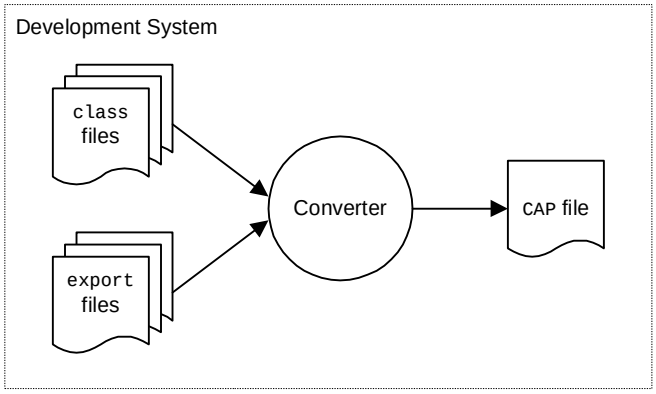

FIGURE 1-1 Java Card Applet Conversion

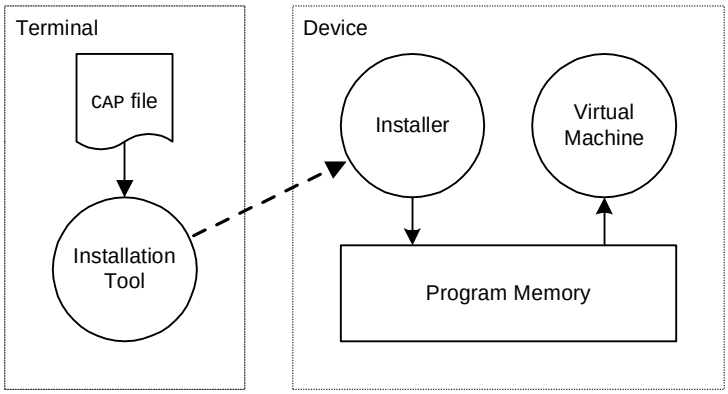

FIGURE 1-2 Java Card Applet Installation

Development of a Java Card applet begins as with any other Java program: a developer writes one or more Java classes, and compiles the source code with a Java compiler, producing one or more `class` files. The applet is run, tested and debugged on a workstation using simulation tools to emulate the device environment. Then, when an applet is ready to be downloaded to a device, the `class` files comprising the applet are converted to a CAP (converted applet) file using a Java Card Converter.

The Java Card Converter takes as input not only the `class` files to be converted, but also one or more `export` files. An `export` file contains name and link information for the contents of other packages that are imported by the classes being converted. When an applet or library package is converted, the converter can also produce an `export` file for that package.

After conversion, the CAP file is copied to a card terminal, such as a desktop computer with a card reader peripheral. Then an installation tool on the terminal loads the CAP file and transmits it to the Java Card technology enabled device. An installation program on the device receives the contents of the CAP file and prepares the applet to be run by the Java Card virtual machine. The virtual machine itself need not load or manipulate CAP files; it need only execute the applet code found in the CAP file that was loaded onto the device by the installation program.

The division of functionality between the Java Card virtual machine and the installation program keeps both the virtual machine and the installation program small. The installation program may be implemented as a Java program and executed on top of the Java Card virtual machine. Since Java Card instructions are denser than typical machine code, this may reduce the size of the installer. The modularity may enable different installers to be used with a single Java Card virtual machine implementation.

1.3 Java Language Security

One of the fundamental features of the Java virtual machine is the strong security provided in part by the `class` file verifier. Many devices that implement the Java Card platform may be too small to support verification of `CAP` files on the device itself. This consideration led to a design that enables verification on a device but does not rely on it. The data in a `CAP` file that is needed only for verification is packaged separately from the data needed for the actual execution of its applet. This allows for flexibility in how security is managed in an implementation.

There are several options for providing language-level security on a Java Card technology enabled device. The conceptually simplest is to verify the contents of a `CAP` file on the device as it is downloaded or after it is downloaded. This option might only be feasible in the largest of devices. However, some subset of verification might be possible even on smaller devices. Other options rely on some combination of one or more of: physical security of the installation terminal, a cryptographically enforced chain of trust from the source of the `CAP` file, and pre-download verification of the contents of a `CAP` file.

The Java Card platform standards say as little as possible about `CAP` file installation and security policies. Since smart cards must serve as secure processors in many different systems with different security requirements, it is necessary to allow a great deal of flexibility to meet the needs of smart card issuers and users.

1.4 Java Card Runtime Environment Security

The standard runtime environment for the Java Card platform is the Java Card Runtime Environment (JCRE). The JCRE consists of an implementation of the Java Card virtual machine along with the Java Card API classes. While the Java Card virtual machine has responsibility for ensuring Java language-level security, the JCRE imposes additional runtime security requirements on devices that implement the JCRE, which results in a need for additional features on the Java Card virtual machine. Throughout this document, these additional features are designated as JCRE-specific.

The basic runtime security feature imposed by the JCRE enforces isolation of applets using what is called an *applet firewall.* The applet firewall prevents the objects that were created by one applet from being used by another applet. This prevents unauthorized access to both the fields and methods of class instances, as well as the length and contents of arrays.

Isolation of applets is an important security feature, but it requires a mechanism to allow applets to share objects in situations where there is a need to interoperate. The JCRE allows such sharing using the concept of shareable interface objects. These objects provide the only way an applet can make its objects available for use by other applets. For more information about using sharable interface objects, see the description of the interface `javacard.framework.Shareable` in the *Java Card 2.1 Application Programming Interface* specification. Some descriptions of firewall-related features will make reference to the `Shareable` interface.

The applet firewall also protects from unauthorized use the objects owned by the JCRE itself. The JCRE can use mechanisms not reflected in the Java Card API to make its objects available for use by applets. A full description of the JCRE-related isolation and sharing features can be found in the *Java Card 2.1 Runtime Environment Specification*.

CHAPTER 2

A Subset of the Java Virtual Machine

This chapter describes the subset of the Java virtual machine and language that is supported in the Java Card 2.1 platform.

2.1 Why a Subset is Needed

It would be ideal if programs for smart cards could be written using all of the Java programming language, but a full implementation of the Java virtual machine is far too large to fit on even the most advanced resource-constrained devices available today.

A typical resource-constrained device has on the order of 1K of RAM, 16K of non-volatile memory (EEPROM or flash) and 24K of ROM. The code for implementing string manipulation, single and double-precision floating point arithmetic, and thread management would be larger than the ROM space on such a device. Even if it could be made to fit, there would be no space left over for class libraries or application code. RAM resources are also very limited. The only workable option is to implement Java Card technology as a subset of the Java platform.

2.2 Java Card Language Subset

Applets written for the Java Card platform are written in the Java programming language. They are compiled using Java compilers. Java Card technology uses a subset of the Java language, and familiarity with the Java platform is required to understand the Java Card platform.

The items discussed in this section are not described to the level of a language specification. For complete documentation on the Java programming language, see *The Java Language Specification* (§1.1).

2.2.1 Unsupported Items

The items listed in this section are elements of the Java programming language and platform that are not supported by the Java Card platform.

2.2.1.1 Unsupported Features

Dynamic Class Loading

Dynamic class loading is not supported in the Java Card platform. An implementation of the Java Card platform is not able to load classes dynamically. Classes are either masked into the card during manufacturing or downloaded through an installation process after the card has been issued. Programs executing on the card may only refer to classes that already exist on the card, since there is no way to download classes during the normal execution of application code.

Security Manager

Security management in the Java Card platform differs significantly from that of the Java platform. In the Java platform, there is a Security Manager class (`java.lang.SecurityManager`) responsible for implementing security features. In the Java Card platform, language security policies are implemented by the virtual machine. There is no Security Manager class that makes policy decisions on whether to allow operations.

Garbage Collection & Finalization

Java Card technology does not require a garbage collector. Nor does Java Card technology allow explicit deallocation of objects, since this would break the Java programming language's required pointer-safety. Therefore, application programmers cannot assume that objects that are allocated are ever deallocated. Storage for unreachable objects will not necessarily be reclaimed.

Finalization is also not required. `finalize()` will not necessarily be called automatically by the Java Card virtual machine, and programmers should not rely on this behavior.

Threads

The Java Card virtual machine does not support multiple threads of control. Java Card programs cannot use class `Thread` or any of the thread-related keywords in the Java programming language.

Cloning

The Java Card platform does not support cloning of objects. Java Card API class `Object` does not implement a `clone` method, and there is no `Cloneable` interface provided.

Access Control in Java Packages

The Java Card language subset supports the package access control defined in the Java language. However, there are two cases that are not supported.

- If a class implements a method with `package` access visibility, a subclass cannot override the method and change the access visibility of the method to `protected` or `public`.
- An interface that is defined with `package` access visibility cannot be extended by an interface with `public` access visibility.

2.2.1.2 Keywords

The following keywords indicate unsupported options related to native methods, threads and memory management.

```
native          synchronized    transient       volatile
```

2.2.1.3 Unsupported Types

The Java Card platform does not support types `char`, `double`, `float` or `long`, or operations on those types. It also does not support arrays of more than one dimension.

2.2.1.4 Classes

In general, none of the Java core API classes are supported in the Java Card platform. Some classes from the `java.lang` package are supported (see §2.2.2.4), but none of the rest are. For example, classes that are *not* supported are `String`, `Thread` (and all thread-related classes), wrapper classes such as `Boolean` and `Integer`, and class `Class`.

System

Class `java.lang.System` is not supported. Java Card technology supplies a class `javacard.framework.JCSystem`, which provides an interface to system behavior.

2.2.2 Supported Items

If a language feature is not explicitly described as unsupported, it is part of the supported subset. Notable supported features are described in this section.

2.2.2.1 Features

Packages

Software written for the Java Card platform follows the standard rules for the Java platform packages. Java Card API classes are written as Java source files, which include package designations. Package mechanisms are used to identify and control access to classes, static fields and static methods. Except as noted in "Access Control in Java Packages" (§2.2.1.1), packages in the Java Card platform are used exactly the way they are in the Java platform.

Dynamic Object Creation

The Java Card platform programs supports dynamically created objects, both class instances and arrays. This is done, as usual, by using the `new` operator. Objects are allocated out of the heap.

As noted in "Garbage Collection & Finalization" (§2.2.1.1), a Java Card virtual machine will not necessarily garbage collect objects. Any object allocated by a virtual machine may continue to exist and consume resources even after it becomes unreachable.

Virtual Methods

Since Java Card objects are Java programming language objects, invoking virtual methods on objects in a program written for the Java Card platform is exactly the same as in a program written for the Java platform. Inheritance is supported, including the use of the `super` keyword.

Interfaces

Java Card classes may define or implement interfaces as in the Java programming language. Invoking methods on interface types works as expected. Type checking and the `instanceof` operator also work correctly with interfaces.

Exceptions

Java Card programs may define, throw and catch exceptions, as in Java programs. Class `Throwable` and its relevant subclasses are supported. (Some `Exception` and `Error` subclasses are omitted, since those exceptions cannot occur in the Java Card platform. See §2.3.3 for specification of errors and exceptions.)

2.2.2.2 Keywords

The following keywords are supported. Their use is the same as in the Java programming language.

`abstract`	`default`	`if`	`private`	`this`
`boolean`	`do`	`implements`	`protected`	`throw`
`break`	`else`	`import`	`public`	`throws`
`byte`	`extends`	`instanceof`	`return`	`try`
`case`	`final`	`int`	`short`	`void`
`catch`	`finally`	`interface`	`static`	`while`
`class`	`for`	`new`	`super`	
`continue`	`goto`	`package`	`switch`	

2.2.2.3 Types

Java programming language types `boolean`, `byte`, `short`, and `int` are supported. Objects (class instances and single-dimensional arrays) are also supported. Arrays can contain the supported primitive data types, objects, and other arrays.

Some Java Card implementations might not support use of the `int` data type. (Refer to §2.2.3.1.)

2.2.2.4 Classes

Most of the classes in the `java.lang` package are not supported in Java Card. The following classes from `java.lang` are supported on the card in a limited form.

Object

Java Card classes descend from `java.lang.Object`, just as in the Java programming language. Most of the methods of `Object` are not available in the Java Card API, but the class itself exists to provide a root for the class hierarchy.

Throwable

Class `Throwable` and its subclasses are supported. Most of the methods of `Throwable` are not available in the Java Card API, but the class itself exists to provide a common ancestor for all exceptions.

2.2.3 Optionally Supported Items

This section describes the optional features of the Java Card platform. An optional feature is not required to be supported in a Java Card compatible implementation. However, if an implementation does include support for an optional feature, it must be supported fully, and exactly as specified in this document.

2.2.3.1 `int`

The `int` keyword and 32-bit integer data types need not be supported in a Java Card implementation. A Java Card virtual machine that does not support the `int` data type will reject programs which use the `int` data type or 32-bit intermediate values.

2.2.4 Limitations of the Java Card Virtual Machine

The limitations of resource-constrained hardware prevent Java Card programs from supporting the full range of functionality of certain Java platform features. The features in question are supported, but a particular virtual machine may limit the range of operation to less than that of the Java platform.

To ensure a level of portability for application code, this section establishes a minimum required level for partial support of these language features.

The limitations here are listed as maximums from the application programmer's perspective. Applets that do not violate these maximum values can be converted into Java Card `CAP` files, and will be portable across all Java Card implementations. From the Java Card virtual machine implementer's perspective, each maximum listed indicates a minimum level of support that will allow portability of applets.

2.2.4.1 Classes

Classes in a Package

A package can contain at most 255 public classes and interfaces.

Interfaces

A class can implement at most 15 interfaces, including interfaces implemented by superclasses.

An interface can inherit from at most 15 superinterfaces.

Static Fields

A class can have at most 256 public or protected static fields.

Static Methods

A class can have at most 256 public or protected static methods.

2.2.4.2 Objects

Methods

A class can implement a maximum of 128 public or protected instance methods, and a maximum of 128 instance methods with package visibility. These limits include inherited methods.

Class Instances

Class instances can contain a maximum of 255 fields, where an `int` data type is counted as occupying two fields.

Arrays

Arrays can hold a maximum of 32767 fields.

2.2.4.3 Methods

The maximum number of local variables that can be used in a method is 255, where an `int` data type is counted as occupying two local variables.

A method can have at most 32767 Java Card virtual machine bytecodes. The number of Java Card bytecodes may differ from the number of Java bytecodes in the Java virtual machine implementation of that method.

2.2.4.4 Switch Statements

The format of the Java Card virtual machine switch instructions limits switch statements to a maximum of 65536 cases. This limit is far greater than the limit imposed by the maximum size of methods (§2.2.4.3).

2.2.4.5 Class Initialization

There is limited support for initialization of static field values in `<clinit>` methods. Static fields of applets may only be initialized to primitive compile-time constant values, or arrays of primitive compile-time constants. Static fields of user libraries may only be initialized to primitive compile-time constant values. Primitive constant data types include `boolean`, `byte`, `short`, and `int`.

2.3 Java Card VM Subset

Java Card technology uses a subset of the Java virtual machine, and familiarity with the Java platform is required to understand the Java Card virtual machine.

The items discussed in this section are not described to the level of a virtual machine specification. For complete documentation on the Java virtual machine, refer to §1.1 of *The Java™ Virtual Machine Specification*.

2.3.1 `class` File Subset

The operation of the Java Card virtual machine can be defined in terms of standard Java platform `class` files. Since the Java Card virtual machine supports only a subset of the behavior of the Java virtual machine, it also supports only a subset of the standard `class` file format.

2.3.1.1 Not Supported in `Class` Files

Field Descriptors

Field descriptors may not contain *BaseType* characters `C`, `D`, `F` or `L`. *ArrayType* descriptors for arrays of more than one dimension may not be used.

Constant Pool

Constant pool table entry tags that indicate unsupported types are not supported.

Constant Type	Value
`CONSTANT_String`	8
`CONSTANT_Float`	4
`CONSTANT_Long`	5
`CONSTANT_Double`	6

TABLE 2-1 Unsupported Java constant pool tags

Constant pool structures for types `CONSTANT_String_info`, `CONSTANT_Float_info`, `CONSTANT_Long_info` and `CONSTANT_Double_info` are not supported.

Fields

In `field_info` structures, the access flags `ACC_VOLATILE` and `ACC_TRANSIENT` are not supported.

Methods

In `method_info` structures, the access flags `ACC_SYNCHRONIZED` and `ACC_NATIVE` are not supported.

2.3.1.2 Supported in Class Files

ClassFile

All items in the `ClassFile` structure are supported.

Field Descriptors

Field descriptors may contain *BaseType* characters `B`, `I`, `S` and `Z`, as well as any *ObjectType*. *ArrayType* descriptors for arrays of a single dimension may also be used.

Method Descriptors

All forms of method descriptors are supported.

Constant pool

Constant pool table entry tags for supported data types are supported.

Constant Type	Value
`CONSTANT_Class`	`7`
`CONSTANT_Fieldref`	`9`
`CONSTANT_Methodref`	`10`
`CONSTANT_InterfaceMethodref`	`11`
`CONSTANT_Integer`	`3`
`CONSTANT_NameAndType`	`12`
`CONSTANT_Utf8`	`1`

TABLE 2-2 Supported Java constant pool tags.

Constant pool structures for types `CONSTANT_Class_info`, `CONSTANT_Fieldref_info`, `CONSTANT_Methodref_info`, `CONSTANT_InterfaceMethodref_info`, `CONSTANT_Integer_info`, `CONSTANT_NameAndType_info` and `CONSTANT_Utf8_info` are supported.

Fields

In `field_info` structures, the supported access flags are `ACC_PUBLIC`, `ACC_PRIVATE`, `ACC_PROTECTED`, `ACC_STATIC` and `ACC_FINAL`.

The remaining components of `field_info` structures are fully supported.

Methods

In `method_info` structures, the supported access flags are `ACC_PUBLIC`, `ACC_PRIVATE`, `ACC_PROTECTED`, `ACC_STATIC`, `ACC_FINAL` and `ACC_ABSTRACT`.

The remaining components of `method_info` structures are fully supported.

Attributes

The `attribute_info` structure is supported. The `Code`, `ConstantValue`, `Exceptions` and `LocalVariableTable` attributes are supported.

2.3.2 Bytecode Subset

The following sections detail the bytecodes that are either supported or unsupported in the Java Card platform. For more details, refer to Chapter 6, “Instruction Set.”

2.3.2.1 Unsupported Bytecodes

lconst_<l>	fconst_<f>	dconst_<d>	ldc2_w2
lload	fload	dload	lload_<n>
fload_<n>	dload_<n>	laload	faload
daload	caload	lstore	fstore
dstore	lstore_<n>	fstore_<n>	dstore_<n>
lastore	fastore	dastore	castore
ladd	fadd	dadd	lsub
fsub	dsub	lmul	fmul
dmul	ldiv	fdiv	ddiv
lrem	frem	drem	lneg
fneg	dneg	lshl	lshr
lushr	land	lor	lxor
i2l	i2f	i2d	l2i
l2f	l2d	f2i	f2d
d2i	d2l	d2f	i2c
lcmp	fcmpl	fcmpg	dcmpl
dcmpg	lreturn	freturn	dreturn
monitorenter	monitorexit	multianewarray	goto_w
jsr_w			

2.3.2.2 Supported Bytecodes

nop	aconst_null	iconst_<i>	bipush
sipush	ldc	ldc_w	iload
aload	iload_<n>	aload_<n>	iaload
aaload	baload	saload	istore
astore	istore_<n>	astore_<n>	iastore
aastore	bastore	sastore	pop
pop2	dup	dup_x1	dup_x2
dup2	dup2_x1	dup2_x2	swap
iadd	isub	imul	idiv
irem	ineg	ior	ishl
ishr	iushr	iand	ixor
iinc	i2b	i2s	if<cond>
ificmp_<cond>	ifacmp_<cond>	goto	jsr
ret	tableswitch	lookupswitch	ireturn
areturn	return	getstatic	putstatic
getfield	putfield	invokevirtual	invokespecial
invokestatic	invokeinterface	new	newarray
anewarray	arraylength	athrow	checkcast
instanceof	wide	ifnull	ifnonnull

2.3.2.3 Static Restrictions on Bytecodes

For it to be acceptable to a Java Card virtual machine, a `class` file must conform to the following restrictions on the static form of bytecodes.

ldc, ldc_w

The *ldc* and *ldc_w* bytecodes can only be used to load integer constants. The constant pool entry at *index* must be a `CONSTANT_Integer` entry. If a program contains an *ldc* or *ldc_w* instruction that is used to load an integer value less than -32768 or greater than 32767, that program will require the optional `int` instructions (§2.2.3.1).

lookupswitch

The value of the *npairs* operand must be less than 65536. The bytecode can contain at most 65535 cases. This limit is far greater than the limit imposed by the maximum size of methods (§2.2.4.3). If a program contains a *lookupswitch* instruction that uses keys of type `int`, that program will require the optional `int` instructions (§2.2.3.1). Otherwise, key values must be in the range -32768 to 32767.

tableswitch

The values of the *high* and *low* operands must both be at least -32768 and at most 32767 (so they can fit in a `short`). The bytecode can contain at most 65536 cases. This limit is far greater than the limit imposed by the maximum size of methods (§2.2.4.3).If a program contains a *tableswitch* instruction that uses indexes of type `int`, that program will require the optional `int` instructions (§2.2.3.1). Otherwise, index values must be in the range -32768 to 32767.

wide

The *wide* bytecode cannot be used to generate local indices greater than 127, and it cannot be used with any instructions other than *iinc*. It can only be used with an *iinc* bytecode to extend the range of the increment constant.

2.3.3 Exceptions

Java Card provides full support for the Java platform's exception mechanism. Users can define, throw and catch exceptions just as in the Java platform. Java Card also makes use of the exceptions and errors defined in *The Java Language Specification* [1]. An updated list of the Java platform's exceptions is provided in the JDK documentation.

Not all of the Java platform's exceptions are supported in Java Card. Exceptions related to unsupported features are naturally not supported. Class loader exceptions (the bulk of the checked exceptions) are not supported. And no exceptions or errors defined in packages other than `java.lang` are supported.

Note that some exceptions may be supported to the extent that their error conditions are detected correctly, but classes for those exceptions will not necessarily be present in the API.

The supported subset is described in the tables below.

2.3.3.1 Uncaught and Uncatchable Exceptions

In the Java platform, uncaught exceptions and errors will cause the virtual machine's current thread to exit. As the Java Card virtual machine is single-threaded, uncaught exceptions or errors will cause the virtual machine to halt. Further response to uncaught exceptions or errors after halting the virtual machine is an implementation-specific policy, and is not mandated in this document.

Some error conditions are known to be unrecoverable at the time they are thrown. Throwing a runtime exception or error that cannot be caught will also cause the virtual machine to halt. As with uncaught exceptions, implementations may take further responses after halting the virtual machine. Uncatchable exceptions and errors which are supported by the Java Card platform may not be reflected in the Java Card API, though the Java Card platform will correctly detect the error condition.

2.3.3.2 Checked Exceptions

Exception	Supported	Not Supported
`ClassNotFoundException`		●
`CloneNotSupportedException`		●
`IllegalAccessException`		●
`InstantiationException`		●
`InterruptedException`		●
`NoSuchFieldException`		●
`NoSuchMethodException`		●

TABLE 2-3 Support of Java checked exceptions

2.3.3.3 Runtime Exceptions

Runtime Exception	Supported	Not Supported
ArithmeticException	●	
ArrayStoreException	●	
ClassCastException	●	
IllegalArgumentException		●
IllegalThreadStateException		●
NumberFormatException		●
IllegalMonitorStateException		●
IllegalStateException		●
IndexOutOfBoundsException	●	
ArrayIndexOutOfBoundsException	●	
StringIndexOutOfBoundsException		●
NegativeArraySizeException	●	
NullPointerException	●	
SecurityException	●	

TABLE 2-4 Support of Java runtime exceptions

2.3.3.4 Errors

Error	Supported	Not Supported
LinkageError	●	
ClassCircularityError	●	
ClassFormatError	●	
ExceptionInInitializerError	●	
IncompatibleClassChangeError	●	
AbstractMethodError	●	
IllegalAccessError	●	
InstantiationError	●	
NoSuchFieldError	●	
NoSuchMethodError	●	
NoClassDefFoundError	●	
UnsatisfiedLinkError	●	
VerifyError	●	
ThreadDeath		●
VirtualMachineError	●	
InternalError	●	
OutOfMemoryError	●	
StackOverflowError	●	
UnknownError	●	

TABLE 2-5 Support of Java errors

CHAPTER 3

Structure of the Java Card Virtual Machine

The specification of the Java Card virtual machine is in many ways quite similar to that of the Java Virtual Machine. This similarity is of course intentional, as the design of the Java Card virtual machine was based on that of the Java Virtual Machine. Rather than reiterate all the details of this specification which are shared with that of the Java Virtual Machine, this chapter will mainly refer to its counterpart in the *Java Virtual Machine Specification, 1st Edition,* providing new information only where the Java Card virtual machine differs.

3.1 Data Types and Values

The Java Card virtual machine supports the same two kinds of data types as the Java Virtual Machine: *primitive types* and *reference types.* Likewise, the same two kinds of values are used: *primitive values* and *reference values.*

The primitive data types supported by the Java Card virtual machine are the *numeric types* and the `returnAddress` type. The numeric types consist only of the *integral types*:

- `byte`, whose values are 8-bit signed two's complement integers
- `short`, whose values are 16-bit signed two's complement integers

Some Java Card virtual machine implementations may also support an additional integral type:

- `int`, whose values are 32-bit signed two's complement integers

Support for `reference` types is identical to that in the Java Virtual Machine.

3.2 Words

The Java Card virtual machine is defined in terms of an abstract storage unit called a *word*. This specification does not mandate the actual size in bits of a word on a specific platform. A word is large enough to hold a value of type `byte`, `short`, `reference` or `returnAddress`. Two words are large enough to hold a value of type `int`.

The actual storage used for values in an implementation is platform-specific. There is enough information present in the descriptor component of a CAP file to allow an implementation to optimize the storage used for values in variables and on the stack.

3.3 Runtime Data Areas

The Java Card virtual machine can support only a single thread of execution. Any runtime data area in the Java Virtual Machine which is duplicated on a per-thread basis will have only one global copy in the Java Card virtual machine.

The Java Card virtual machine's heap is not required to be garbage collected. Objects allocated from the heap will not necessarily be reclaimed.

This specification does not include support for `native` methods, so there are no native method stacks.

Otherwise, the runtime data areas are as documented for the Java Virtual Machine.

3.4 Contexts

Each applet running on a Java Card virtual machine is associated with an execution *context*. The Java Card virtual machine uses the context of the current frame to enforce security policies for inter-applet operations.

There is a one-to-one mapping between contexts and packages in which applets are defined. An easy way to think of a context is as the runtime equivalent of a package, since Java packages are compile-time constructs and have no direct representation at runtime. As a consequence, all applets managed by applet instances of applet classes from the same package will share the same context.

The Java Card Runtime Environment also has its own context. Framework objects execute in this *JCRE context*.

The context of the currently executing method is known as the *current context*. Every object in a Java Card virtual machine is owned by a particular context. The *owning context* is the context that was current when the object was created.

When a method in one context successfully invokes a method on an object in another context, the Java Card virtual machine performs a *context switch*. Afterwards the invoked method's context becomes the current context. When the invoked method returns, the current context is switched back to the previous context.

3.5 Frames

Java Card virtual machine *frames* are very similar to those defined for the Java Virtual Machine. Each frame has a set of local variables and an operand stack. Frames also contain a reference to a constant pool, but since all constant pools for all classes in a package are merged, the reference is to the constant pool for the current class' package.

Each frame also includes a reference to the context in which the current method is executing.

3.6 Representation of Objects

The Java Card virtual machine does not mandate a particular internal structure for objects or a particular layout of their contents. However, the core components in a CAP file are defined assuming a default structure for certain runtime structures (such as descriptions of classes), and a default layout for the contents of dynamically allocated objects. Information from the descriptor component of the CAP file can be used to format objects in whatever way an implementation requires.

3.7 Special Initialization Methods

The Java Card virtual machine supports *instance initialization methods* exactly as does the Java Virtual Machine.

The Java Card virtual machine includes only limited support for *class or interface initialization methods*. There is no general mechanism for executing `<clinit>` methods on a Java Card virtual machine. Instead, a CAP file includes information for initializing class data as defined in Chapter 2, “A Subset of the Java Virtual Machine.”

3.8 Exceptions

Exception support in the Java Card virtual machine is identical to support for exceptions in the Java Virtual Machine.

3.9 Binary File Formats

This specification defines two binary file formats which enable platform-independent development, distribution and execution of Java Card software.

The CAP file format describes files that contain executable code and can be downloaded and installed onto a Java Card enabled device. A CAP file is produced by a Java Card Converter tool, and contains a converted form of an entire package of Java classes. This file format's relationship to the Java Card virtual machine is analogous to the relationship of the `class` file format to the Java Virtual Machine.

The `export` file format describes files that contain the public linking information of Java Card packages. A package's `export` file is used when converting client packages of that package.

3.10 Instruction Set Summary

The Java Card virtual machine instruction set is quite similar to the Java Virtual Machine instruction set. Individual instructions consist of a one-byte *opcode* and zero or more *operands*. The pseudo-code for the Java Card virtual machine's instruction fetch-decode-execute loop is the same. Multi-byte operand data is also encoded in *big-endian* order.

There are a number of ways in which the Java Card virtual machine instruction set diverges from that of the Java Virtual Machine. Most of the differences are due to the Java Card virtual machine's more limited support for data types. Another source of

divergence is that the Java Card virtual machine is intended to run on 8-bit and 16-bit architectures, whereas the Java Virtual Machine was designed for a 32-bit architecture. The rest of the differences are all oriented in one way or another toward optimizing the size or performance of either the Java Card virtual machine or Java Card programs. These changes include inlining constant pool data directly in instruction opcodes or operands, adding multiple versions of a particular instruction to deal with different datatypes, and creating composite instructions for operations on the current object.

3.10.1 Types and the Java Card Virtual Machine

The Java Card virtual machine supports only a subset of the types supported by the Java Virtual Machine. This subset is described in Chapter 2, "A Subset of the Java Virtual Machine." Type support is reflected in the instruction set, as instructions encode the data types on which they operate.

Given that the Java Card virtual machine supports fewer types than the Java Virtual Machine, there is an opportunity for better support for smaller data types. Lack of support for large numeric data types frees up space in the instruction set. This extra instruction space has been used to directly support arithmetic operations on the `short` data type.

Some of the extra instruction space has also been used to optimize common operations. Type information is directly encoded in field access instructions, rather than being obtained from an entry in the constant pool.

TABLE 3-1 summarizes the type support in the instruction set of the Java Card virtual machine. Only instructions that exist for multiple types are listed. Wide and composite forms of instructions are not listed either. A specific instruction, with type information, is built by replacing the *T* in the instruction template in the opcode column by the letter representing the type in the type column. If the type column for some instruction is blank, then no instruction exists supporting that operation on that type. For instance, there is a load instruction for type `short`, *sload*, but there is no load instruction for type `byte`.

opcode	byte	short	int	reference
Tspush	*bspush*	*sspush*		
Tipush	*bipush*	*sipush*	*iipush*	
Tconst		*sconst*	*iconst*	*aconst*
Tload		*sload*	*iload*	*aload*
Tstore		*sstore*	*istore*	*astore*
Tinc		*sinc*	*iinc*	
Taload	*baload*	*saload*	*iaload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*aastore*
Tadd		*sadd*	*iadd*	
Tsub		*ssub*	*isub*	
Tmul		*smul*	*imul*	
Tdiv		*sdiv*	*idiv*	
Trem		*srem*	*irem*	
Tneg		*sneg*	*ineg*	
Tshl		*sshl*	*ishl*	
Tshr		*sshr*	*ishr*	
Tushr		*sushr*	*iushr*	
Tand		*sand*	*iand*	
Tor		*sor*	*ior*	
Txor		*sxor*	*ixor*	
s2T	*s2b*		*s2i*	
i2T	*i2b*	*i2s*		
Tcmp			*icmp*	
if_TcmpOP		*if_scmpOP*		*if_acmpOP*
Tlookupswitch		*slookupswitch*	*ilookupswitch*	
Ttableswitch		*stableswitch*	*itableswitch*	
Treturn		*sreturn*	*ireturn*	*areturn*
getstatic_T	*getstatic_b*	*getstatic_s*	*getstatic_i*	*getstatic_a*
putstatic_T	*putstatic_b*	*putstatic_s*	*putstatic_i*	*putstatic_a*
getfield_T	*getfield_b*	*getfield_s*	*getfield_i*	*getfield_a*
putfield_T	*putfield_b*	*putfield_s*	*putfield_i*	*putfield_a*

TABLE 3-1 Type support in the Java Card Virtual Machine Instruction Set

The mapping between Java storage types and Java Card virtual machine computational types is summarized in TABLE 3-2.

Java (Storage) Type	Size in Bits	Computational Type
`byte`	8	`short`
`short`	16	`short`
`int`	32	`int`

TABLE 3-2 Storage types and computational types

Chapter 7, “Java Card Virtual Machine Instruction Set,” describes the Java Card virtual machine instruction set in detail.

CHAPTER 4

Binary Representation

This chapter presents information about the binary representation of Java Card programs. Java Card binaries are usually contained in files, therefore this chapter addresses binary representation in terms of this common case.

Several topics relating to binary representation are covered. The first section describes the basic organization of program representation in `export` and `CAP` files, as well as the use of the JAR file containers. The second section covers how Java Card applets and packages are named using unique identifiers. The third section presents the scheme used for naming and linking items within Java Card packages. The fourth and fifth sections describe the constraints for upward compatibility between different versions of a Java Card binary program file, and versions assigned based upon that compatibility.

4.1 Java Card File Formats

Java programs are represented in compiled, binary form as `class` files. Java `class` files are used not only to execute programs on a Java virtual machine, but also to provide type and name information to a Java compiler. In the latter role, a `class` file is essentially used to document the API of its class to client code. That client code is compiled into its own `class` file, including symbolic references used to dynamically link to the API class at runtime.

Java Card technology uses a different strategy for binary representation of programs. Executable binaries and interface binaries are represented in two separate files. These files are respectively called `CAP` files (for converted applet) and `export` files.

4.1.1 `Export` File Format

`Export` files are not used directly on a device that implements a Java Card virtual machine. However, the information in an `export` file is critical to the operation of the virtual machine on a device. An `export` file can be produced by a Java Card converter when a package is converted. This package's `export` file can be used later to convert another package that imports classes from the first package. Information in the `export` file is included in the `CAP` file of the second package, then is used on the device to link the contents of the second package to items imported from the first package.

A Java Card `export` file contains the public interface information for an entire package of classes. This means that an `export` file only contains information about the public API of a package, and does not include information used to link classes within a package.

The name of an `export` file is the last portion of the package specification followed by the extension '`.exp`'. For example, the name of the `export` file of the `javacard.framework` package must be `framework.exp`. Operating systems that impose limitations on file name lengths may transform an `export` file's name according to their own conventions.

For a complete description of the Java Card `export` file format, see Chapter 5.

4.1.2 `CAP` File Format

A Java Card `CAP` file contains a binary representation of a package of classes that can be installed on a device and used to execute the package's classes on a Java Card virtual machine.

A `CAP` file is produced by a Java Card converter when a package of classes is converted. A `CAP` file can contain a user library, or one or more applet definitions. A `CAP` file consists of a set of components, each of which describes a different aspect of the contents. The set of components in a `CAP` file can vary, depending on whether the file contains a library or applet definition(s).

For a complete description of the Java Card `CAP` File format, see Chapter 6.

4.1.3 JAR File Container

The JAR file format is used as the container format for `CAP` files. What this specification calls a "`CAP` file" is just a JAR file that contains the required set of `CAP` components (see Chapter 6).

CAP component files in a JAR file are located in a subdirectory called javacard that is in a directory representing the package. For example, the CAP component files of the package com.sun.framework are located in the directory com/sun/framework/javacard.

An export file may also be contained in a JAR file, whether that JAR file contains CAP component files or not. If an export file is included, it must be located in the same directory as the CAP component files for that package would be.

The name of a JAR file containing CAP component files is not defined as part of this specification. Other files, including other CAP files, may also reside in a JAR file that contains CAP component files.

4.2 AID-based Naming

This section describes the mechanism used for naming applets and packages in Java Card CAP files and export files, and custom components in Java Card CAP files. Java class files use Unicode strings to name Java packages. As the Java Card platform does not include support for strings, an alternative mechanism for naming is provided.

ISO 7816 is a multipart standard that describes a broad range of technology for building smart card systems. ISO 7816-5 defines the AID (application identifier) data format to be used for unique identification of card applications (and certain kinds of files in card file systems). The Java Card platform uses the AID data format to identify applets and packages. AIDs are administered by the International Standards Organization (ISO), so they can be used as unique identifiers.

4.2.1 The AID Format

This section presents a minimal description of the AID data format used in Java Card technology. For complete details, refer to ISO 7816-5, AID Registration Category ‘D’ format.

The AID format used by the Java Card platform is an array of bytes that can be interpreted as two distinct pieces, as shown in FIGURE 4-1. The first piece is a 5-byte value known as a RID (resource identifier). The second piece is a variable length value known as a PIX (proprietary identifier extension). A PIX can be from 0 to 11 bytes in length. Thus an AID can be from 5 to 16 bytes in total length.

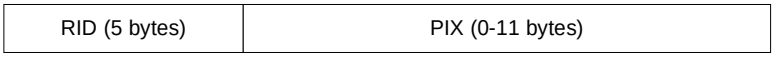

FIGURE 4-1 AID Format

ISO controls the assignment of RIDs to companies, with each company obtaining its own unique RID from the ISO. Companies manage assignment of PIXs for AIDs using their own RIDs.

4.2.2 AID Usage

In the Java platform, packages are uniquely identified using Unicode strings and a naming scheme based on internet domain names. In the Java Card platform, packages and applets are identified using AIDs.

Any package that is represented in an `export` file must be assigned a unique AID. The AID for a package is constructed from the concatenation of the company's RID and a PIX for that package. This AID corresponds to the string name for the package, as shown in FIGURE 4-2.

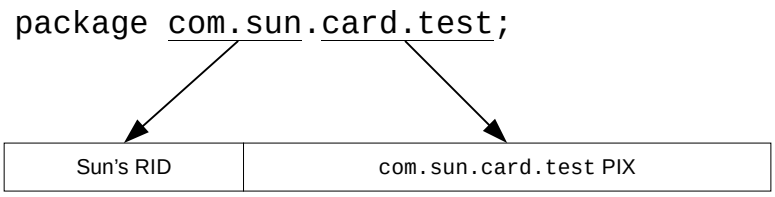

FIGURE 4-2 Mapping package identifiers to AIDs

Each applet installed on a Java Card technology enabled device must also have a unique AID. This AID is constructed similarly to a package AID. It is a concatenation of the applet provider's RID and PIX for that applet. An applet AID must not have the same value as the AID of any package or the AID of any other applet. If a `CAP` file defines multiple applets, all applet AIDs in that `CAP` file must have the same RID.

Custom components defined in a `CAP` file are also identified using AIDs. Like AIDs for applets and packages, component AIDs are formed by concatenating a RID and a PIX. All AIDs of new components must have the same RID as the AID for the package defined in the `CAP` file.

4.3 Token-based Linking

This section describes a scheme that allows downloaded software to be linked against APIs on a Java Card technology enabled device. The scheme represents referenced items as opaque tokens, instead of Unicode strings as are used in Java class files. The two basic requirements of this linking scheme are that it allows linking on the device, and that it does not require internal implementation details of APIs to be revealed to clients of those APIs. Secondary requirements are that the scheme be efficient in terms of resource use on the device, and have acceptable performance for linking. And of course, it must preserve the semantics of the Java language.

4.3.1 Externally Visible Items

Classes (including Interfaces) in Java packages may be declared with public or package visibility. A class's methods and fields may be declared with public, protected, package or private visibility. For purposes of this document, we define public classes, public or protected fields, and public or protected methods to be *externally visible* from the package. All externally visible items are described in a package's `export` file.

Each externally visible item must have a token associated with it to enable references from other packages to the item to be resolved on a device. There are six kinds of items in a package that require external identification.

- Classes (including Interfaces)
- Static Fields
- Static Methods
- Instance Fields
- Virtual Methods
- Interface Methods

4.3.2 Private Tokens

Items that are not externally visible are *internally visible.* Internally visible items are not described in a package's `export` file, but some such items use *private tokens* to represent internal references. External references are represented by *public tokens.* There are two kinds of items that can be assigned private tokens.

- Instance Fields
- Virtual Methods

4.3.3 The `Export` File and Conversion

Each externally visible item in a package has an entry in the package's `export` file. Each entry holds the item's name and its token. Some entries may include additional information as well. For detailed information on the `export` file format, see Chapter 5, "The Export File Format."

The `export` file is used to map names for imported items to tokens during package conversion. The Java Card converter uses these tokens to represent references to items in an imported package.

For example, during the conversion of the class files of applet A, the `export` file of `javacard.framework` is used to find tokens for items in the API that are used by the applet. Applet A creates a new instance of framework class `OwnerPIN`. The framework `export` file contains an entry for `javacard.framework.OwnerPIN` that holds the token for this class. The converter places this token in the `CAP` file's constant pool to represent an unresolved reference to the class. The token value is later used to resolve the reference on a device.

4.3.4 References – External and Internal

In the context of a `CAP` file, references to items are made indirectly through a package's constant pool. References to items in other packages are called *external*, and are represented in terms of tokens. References to items in the same `CAP` file are called *internal*, and are represented either in terms of tokens, or in a different internal format.

An external reference to a class is composed of a package token and a class token. Together those tokens specify a certain class in a certain package. An internal reference to a class is a 15-bit value that is a pointer to the class structure's location within the `CAP` file.

An external reference to a static class member, either a field or method, consists of a package token, a class token, and a token for the static field or static method. An internal reference to a static class member is a 16-bit value that is a pointer to the item's location in the `CAP` file.

References to instance fields, virtual methods and interface methods consist of a class reference and a token of the appropriate type. The class reference determines whether the reference is external or internal.

4.3.5 Installation and Linking

External references in a `CAP` file can be resolved on a device from token form into the internal representation used by the virtual machine.

A token can only be resolved in the context of the package that defines it. Just as the `export` file maps from a package's externally visible names to tokens, there is a set of link information for each package on the device that maps from tokens to resolved references.

4.3.6 Token Assignment

Tokens for an API are assigned by the API's developer and published in the package `export` file(s) for that API. Since the name-to-token mappings are published, an API developer may choose any order for tokens (subject to the constraints listed below).

A particular device platform can resolve tokens into whatever internal representation is most useful for that implementation of a Java Card virtual machine. Some tokens may be resolved to indices. For example, an instance field token may be resolved to an index into a class instance's fields. In such cases, the token value is distinct from and unrelated to the value of the resolved index.

4.3.7 Token Details

Each kind of item in a package has its own independent scope for tokens of that kind. The token range and assignment rules for each kind are listed in TABLE 4-1.

Token Type	Range	Type	Scope
Package	0 - 127	Private	`CAP` File
Class	0 - 255	Public	Package
Static Field	0 - 255	Public	Class
Static Method	0 - 255	Public	Class
Instance Field	0 - 255	Public or Private	Class
Virtual Method	0 - 127	Public or Private	Class Hierarchy
Interface Method	0 - 127	Public	Class

TABLE 4-1 Token Range, Type and Scope

4.3.7.1 Package

All package references from within a `CAP` file are assigned private *package tokens*; package tokens will never appear in an `export` file. Package token values must be in the range from 0 to 127, inclusive. The tokens for all the packages referenced from classes in a `CAP` file are numbered consecutively starting at zero. The ordering of package tokens is not specified.

4.3.7.2 Classes and Interfaces

All externally visible classes in a package are assigned public *class tokens*. Package-visible classes are not assigned tokens. Class token values must be in the range from 0 to 255, inclusive. The tokens for all the public classes in a package are numbered consecutively starting at zero. The ordering of class tokens is not specified.

4.3.7.3 Static Fields

All externally visible static fields in a package are assigned public *static field tokens*. Package-visible and private static fields are not assigned tokens. No tokens are assigned for final static fields that are initialized to primitive, compile-time constants, as these fields are never linked on a device. The tokens for all other externally visible static fields in a class are numbered consecutively starting at zero. Static fields token values must be in the range from 0 to 255, inclusive. The ordering of static field tokens is not specified.

4.3.7.4 Static Methods

All externally visible static methods in a package are assigned public *static method tokens*, including statically bound instance methods. Static method token values must be in the range from 0 to 255, inclusive. Package-visible and private static methods are not assigned tokens. The tokens for all the externally visible static methods in a class are numbered consecutively starting at zero. The ordering of static method tokens is not specified.

4.3.7.5 Instance Fields

All instance fields defined in a package are assigned either public or private *instance field tokens*. Instance field token values must be in the range from 0 to 255, inclusive. Public and private tokens for instance fields are assigned from the same namespace. The tokens for all the instance fields in a class are numbered consecutively starting at zero, except that the token after an `int` field is skipped and the token for the following field is numbered two greater than the token of the `int` field. Tokens for

externally visible fields must be numbered less than the tokens for package and private fields. For public tokens, the tokens for reference type fields must be numbered greater than the tokens for primitive type fields. For private tokens, the tokens for reference type fields must be numbered less than the tokens for primitive type fields. Beyond that the ordering of instance field tokens in a class is not specified.

Visibility	Category	Type	Token Value
public and protected fields (public tokens)	primitive	`boolean`	0
		`byte`	1
		`short`	2
	references	`byte[]`	3
		`Applet`	4
package and private fields (private tokens)	references	`short[]`	5
		`Object`	6
	primitive	`int`	7
		`short`	9

FIGURE 4-3 Tokens for Instance Fields

4.3.7.6 Virtual Methods

All virtual methods defined in a package are assigned either public or private *virtual method tokens*. Virtual method token values must be in the range from 0 to 127, inclusive. Public and private tokens for virtual methods are assigned from different namespaces. The high bit of the byte containing a virtual method token is set to one if the token is a private token.

Public tokens for the externally visible introduced virtual methods in a class are numbered consecutively starting at one greater than the highest numbered public virtual method token of the class's superclass. If a method overrides a method implemented in the class's superclass, that method uses the same token number as the method in the superclass. The high bit of the byte containing a public virtual method token is always set to zero, to indicate it is a public token. The ordering of public virtual method tokens in a class is not specified.

Private virtual method tokens are assigned differently from public virtual method tokens. If a class and its superclass are defined in the same package, the tokens for the package-visible introduced virtual methods in that class are numbered consecutively starting at one greater than the highest numbered private virtual method token of the class's superclass. If the class and its superclass are defined in different packages, the tokens for the package-visible introduced virtual methods in that class are numbered consecutively starting at zero. If a method overrides a method implemented in the class's superclass, that method uses the same token

number as the method in the superclass. The definition of the Java programming language specifies that overriding a package-visible virtual method is only possible if both the class and its superclass are defined in the same package. The high bit of the byte containing a virtual method token is always set to one, to indicate it is a private token. The ordering of private virtual method tokens in a class is not specified.

4.3.7.7 Interface Methods

All interface methods defined in a package are assigned public *interface method tokens*, as interface methods are always public. Interface methods tokens values must be in the range from 0 to 127, inclusive. The tokens for all the interface methods defined in or inherited by an interface are numbered consecutively starting at zero. The token value for an interface method in a given interface is unrelated to the token values of that same method in any of the interface's superinterfaces. The high bit of the byte containing an interface method token is always set to zero, to indicate it is a public token. The ordering of interface method tokens is not specified.

4.4 Binary Compatibility

In the Java programming language the granularity of binary compatibility can be between classes since binaries are stored in individual `class` files. In Java Card systems Java packages are processed as a single unit, and therefore the granularity of binary compatibility is between packages. In Java Card systems the *binary* of a package is represented in a `CAP` file, and the API of a package is represented in an `export` file.

In a Java Card system, a change to a type in a Java package results in a new `CAP` file. A new `CAP` file is *binary compatible with* (equivalently, does not *break compatibility with*) a preexisting `CAP` file if another `CAP` file converted using the `export` file of the preexisting `CAP` file can link with the new `CAP` file without errors.

FIGURE 4-4 shows an example of binary compatible `CAP` files, `p1` and `p1'`. The preconditions for the example are: the package `p1` is converted to create the `p1 CAP` file and `p1 export` file, and package `p1` is modified and converted to create the `p1' CAP` file. Package `p2` imports package `p1`, and therefore when the `p2 CAP` file is

created the `export` file of `p1` is used. In the example, `p2` is converted using the original `p1` `export` file. Because `p1'` is binary compatible with `p1`, `p2` may be linked with either the `p1` `CAP` file or the `p1'` `CAP` file.

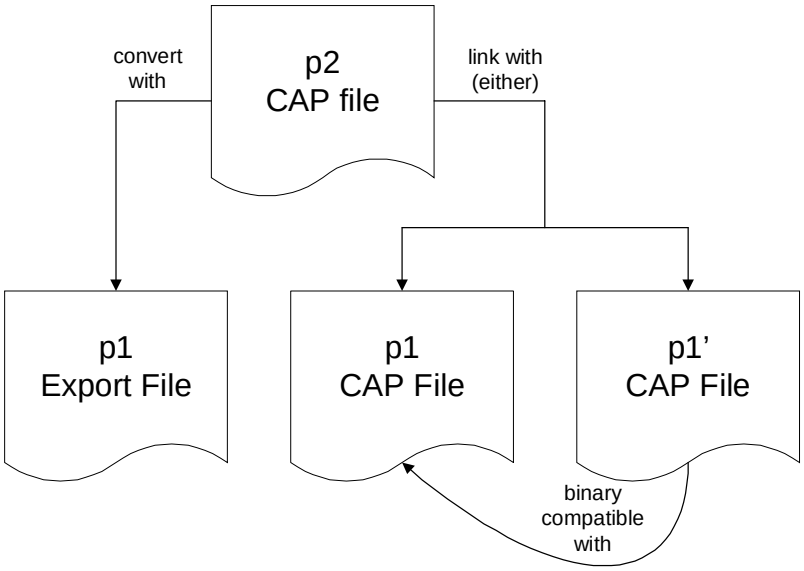

FIGURE 4-4 Binary compatibility example

Any modification that causes binary incompatibility in the Java programming language also causes binary incompatibility in Java Card systems. These modifications are described as causing a potential error in *The Java™ Language Specification*. Any modification that does not cause binary incompatibility in the Java programming language does not cause binary incompatibility in a Java Card system, except under the following conditions:

- the value of a token assigned to an element in the API of a package is changed;
- the value of an externally visible `final` `static` field (compile-time constant) is changed;
- an externally visible virtual method that does not override a preexisting method is added to a non-`final` `public` class.

Tokens are used to resolve references to imported elements of a package. If a token value is modified, a linker on a device is unable to associate the new token value with the previous token value of the element, and therefore is unable to resolve the reference correctly.

Compile-time constants are not stored as fields in `CAP` files. Instead their values are recorded in `export` files and placed inline in the bytecodes in `CAP` files. These values are said to be pre-linked in a `CAP` file of a package that imports those constants.

During execution, information is not available to determine whether the value of an inlined constant is the same as the value defined by the binary of the imported package.

As described above, tokens assigned to `public` and `protected` virtual methods are scoped to the hierarchy of a class. Tokens assigned to `public` and `protected` virtual methods introduced in a subclass have values starting at one greater than the maximum token value assigned in a superclass. If a new, non-override, `public` or `protected` virtual method is introduced in a superclass it is assigned a token value that would otherwise have been assigned in a subclass. Therefore, two unique virtual methods could be assigned the same token value within the same class hierarchy, making resolution of a reference to one of the methods ambiguous.

4.5 Package Versions

Each implementation of a package in a Java Card system is assigned a pair of major and minor version numbers. These version numbers are used to indicate binary compatibility or incompatibility between successive implementations of a package.

4.5.1 Assigning

The major and minor versions of a package are assigned by the package provider. It is recommended that the initial implementation of a package be assigned a major version of 1 and a minor version of 0. However, any values may be chosen. It is also recommended that when either a major or a minor version is incremented, it is incremented exactly by 1.

A major version must be changed when a new implementation of a package is not binary compatible with the previous implementation. The value of the new major version must be greater than the major version of the previous implementation. When a major version is changed, the associated minor version must be assigned the value of 0.

When a new implementation of a package is binary compatible with the previous implementation, it must be assigned a major version equal to the major version of the previous implementation. The minor version assigned to the new implementation must be greater than the minor version of the previous implementation.

4.5.2 Linking

Both an `export` file and a `CAP` file contain the major and minor version numbers of the package described. When a `CAP` file is installed on a Java Card enabled device a *resident image* of the package is created, and the major and minor version numbers are recorded as part of that image. When an `export` file is used during preparation of a `CAP` file, the version numbers indicated in the `export` file are recorded in the `CAP` file.

During installation, references from the package of the `CAP` file being installed to an imported package can be resolved only when the version numbers indicated in the `export` file used during preparation of the `CAP` file are compatible with the version numbers of the resident image. They are compatible when the major version numbers are equal and the minor version of the `export` file is less than or equal to the minor version of the resident image.

CHAPTER 5

The `Export` File Format

This chapter describes the Java Card virtual machine `export` file format. Compliant Java Card Converters must be capable of producing and consuming all `export` files that conform to the specification provided in this chapter. (Refer to Chapter 4, “Binary Representation.”)

An `export` file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two and four consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high-order bytes come first.

This chapter defines its own set of data types representing Java Card `export` file data: The types `u1`, `u2`, and `u4` represent an unsigned one-, two-, and four-byte quantities, respectively.

The Java Card `export` file format is presented using pseudo structures written in a C-like structure notation. To avoid confusion with the fields of Java Card virtual machine classes and class instances, the contents of the structures describing the Java Card `export` file format are referred to as *items*. Unlike the fields of a C structure, successive items are stored in the Java Card file sequentially, without padding or alignment.

Variable-sized *tables*, consisting of variable-sized items, are used in several `export` file structures. Although we will use C-like array syntax to refer to table items, the fact that tables are streams of varying-sized structures means that it is not possible to directly translate a table index into a byte offset into the table.

In a data structure that is referred to as an *array*, the elements are equal in size.

5.1 `Export` File Name

As described in §4.1.1, the name of a `export` file must be the last portion of the package specification followed by the extension '`.exp`'. For example, the name of the `export` file of the `javacard.framework` package must be `framework.exp`. Operating systems that impose limitations on file name lengths may transform an `export` file's name according to its conventions.

5.2 Containment in a Jar File

As described in §4.1.3, Java Card `CAP` files are contained in a JAR file. If an `export` file is also stored in a JAR file, it must also be located in a directory called `javacard` that is a subdirectory package's directory. For example, the `framework.exp` file would be located in the subdirectory `javacard/framework/javacard`.

5.3 `Export` File

An `export` file is defined by the following structure:

```
ExportFile {
       u4 magic
       u1 minor_version
       u1 major_version
       u2 constant_pool_count
       cp_info constant_pool[constant_pool_count]
       u2 this_package
       u1 export_class_count
       class_info classes[export_class_count]
}
```

The items in the `ExportFile` structure are as follows:

`magic`

The `magic` item contains the magic number identifying the `ExportFile` format; it has the value 0x00FACADE.

`minor_version, major_version`

The `minor_version` and `major_version` items are the minor and major version numbers of this `export` file. An implementation of a Java Card virtual machine supports `export` files having a given major version number and minor version numbers in the range 0 through some particular `minor_version`.

If a Java Card virtual machine encounters an `export` file with the supported major version but an unsupported minor version, the Java Card virtual machine must not attempt to interpret the content of the `export` file. However, it will be feasible to upgrade a Java Card virtual machine to support the newer minor version.

A Java Card virtual machine must not attempt to interpret a `export` file with a different major version. A change of the major version number indicates a major incompatibility change, one that requires a fundamentally different Java Card virtual machine.

In this specification, the major version of the `export` file has the value 2 and the minor version has the value 1. Only Sun Microsystems, Inc. may define the meaning and values of new `export` file versions.

`constant_pool_count`

The `constant_pool_count` item is a non-zero, positive value that indicates the number of constants in the constant pool.

`constant_pool[]`

The `constant_pool` is a table of variable-length structures representing various string constants, class names, field names and other constants referred to within the `ExportFile` structure.

Each of the `constant_pool` table entries, including entry zero, is a variable-length structure whose format is indicated by its first “tag” byte.

There are no ordering constrains on entries in the `constant_pool` table.

`this_package`

The value of `this_package` must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Package_info` (§5.4.1) structure representing the package defined by this `ExportFile`.

`export_class_count`

The value of the `export_class_count` item gives the number of elements in the `classes` table.

`classes[]`

Each value of the `classes` table is a variable-length `class_info` structure (§5.5) giving the description of a publicly accessible class or interface declared in this package. If the `ACC_LIBRARY` flag item in the `CONSTANT_Package_info` (§5.4.1) structure indicated by the `this_package` item is set, the `classes` table has an entry for each `public` class and interface declared in this package. If the `ACC_LIBRARY` flag item is not set, the `classes` table has an entry for each shareable interface declared in this package.[1]

5.4 Constant Pool

All `constant_pool` table entries have the following general format:

```
cp_info {
       u1 tag
       u1 info[]
}
```

Each item in the `constant_pool` must begin with a 1-byte `tag` indicating the kind of `cp_info` entry. The content of the `info` array varies with the value of `tag`. The valid tags and their values are listed in TABLE 5-1. Each tag byte must be followed by two or more bytes giving information about the specific constant. The format of the additional information varies with the tag value.

Constant Type	Value
`CONSTANT_Package`	13
`CONSTANT_Interfaceref`	7
`CONSTANT_Integer`	3
`CONSTANT_Utf8`	1

TABLE 5-1 Export file constant pool tags

1. This restriction of exporting only shareable interfaces in non-library packages is imposed by the firewall defined in the Java Card™ Runtime Environment (JCRE) 2.1 Specification.

5.4.1 CONSTANT_Package

The `CONSTANT_Package_info` structure is used to represent a package:

```
CONSTANT_Package_info {
       u1 tag
       u1 flags
       u2 name_index
       u1 minor_version
       u1 major_version
       u1 aid_length
       u1 aid[aid_length]
}
```

The items of the `CONSTANT_Package_info` structure are the following:

`tag`

The `tag` item has the value of `CONSTANT_Package` (13).

`flags`

The `flags` item is a mask of modifiers that apply to this package. The `flags` modifiers are shown in the following table.

Flags	Value
ACC_LIBRARY	0x01

TABLE 5-2 Export file package flags

The `ACC_LIBRARY` flag has the value of one if this package does not define and declare any applets. In this case it is called a *library package*. Otherwise `ACC_LIBRARY` has the value of zero.

If the package is not a library package this `export` file can only contain shareable interfaces.[1] A shareable interface is either the `javacard.framework.Shareable` interface or an interface that extends the `javacard.framework.Shareable` interface.

All other flag values are reserved by the Java Card virtual machine. Their values must be zero.

1. This restriction is imposed by the firewall defined in the Java Card™ Runtime Environment (JCRE) 2.1 Specification.

`name_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§5.4.4) structure representing a valid Java package name.

As in Java class files, ASCII periods (‘.’) that normally separate the identifiers in a package name are replaced by ASCII forward slashes (‘/’). For example, the package name `javacard.framework` is represented in a `CONSTANT_Utf8_info` structure as `javacard/framework`.

`minor_version, major_version`

The `minor_version` and `major_version` items are the minor and major version numbers of this package. These values uniquely identify the particular implementation of this package and indicate the binary compatibility between packages. See §4.5 for a description of assigining and using package version numbers.

`aid_length`

The value of the `aid_length` item gives the number of bytes in the `aid` array. Valid values are between 5 and 16, inclusive.

`aid[]`

The `aid` array contains the ISO AID of this package (§4.2).

5.4.2 CONSTANT_Interfaceref

The `CONSTANT_Interfaceref_info` structure is used to represent an interface:

```
CONSTANT_Interfaceref_info {
       u1 tag
       u2 name_index
}
```

The items of the `CONSTANT_Interfaceref_info` structure are the following:

`tag`

The `tag` item has the value of `CONSTANT_Interface` (7).

`name_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§5.4.4) structure representing a valid fully qualified Java interface name. These names are fully qualified because they may be defined in a package other than the one described in the `export` file.

As in Java class files, ASCII periods (‘.’) that normally separate the identifiers in a class or interface name are replaced by ASCII forward slashes (‘/’). For example, the interface name `javacard.framework.Shareable` is represented in a `CONSTANT_Utf8_info` structure as `javacard/framework/Shareable`.

5.4.3 CONSTANT_Integer

The `CONSTANT_Integer_info` structure is used to represent four-byte numeric (int) constants:

```
CONSTANT_Integer_info {
       u1 tag
       u4 bytes
}
```

The items of the CONSTANT_Integer_info structure are the following:

`tag`

The `tag` item has the value of `CONSTANT_Integer` (3).

`bytes`

The `bytes` item of the `CONSTANT_Integer_info` structure contains the value of the `int` constant. The bytes of the value are stored in big-endian (high byte first) order.

5.4.4 CONSTANT_Utf8

The `CONSTANT_Utf8_info` structure is used to represent constant string values. UTF-8 strings are encoded in the same way as described in *The Java™ Virtual Machine Specification* (§ 4.4.7).

The `CONSTANT_Utf8_info` structure is:

```
CONSTANT_Utf8_info {
       u1 tag
       u2 length
       u1 bytes[length]
}
```

The items of the `CONSTANT_Utf8_info` structure are the following:

`tag`

The `tag` item has the value of `CONSTANT_Utf8` (1).

`length`

The value of the `length` item gives the number of bytes in the `bytes` array (not the length of the resulting string). The strings in the `CONSTANT_Utf8_info` structure are not null-terminated.

`bytes[]`

The `bytes` array contains the bytes of the string. No byte may have the value `(byte)0` or `(byte)0xF0-(byte)0xFF`.

5.5 Classes and Interfaces

Each class and interface is described by a variable-length `class_info` structure. The format of this structure is:

```
class_info {
       u1 token
       u2 access_flags
       u2 name_index
       u2 export_interfaces_count
       u2 interfaces[export_interfaces_count]
       u2 export_fields_count
       field_info fields[export_fields_count]
       u2 export_methods_count
       method_info methods[export_methods_count]
}
```

The items of the `class_info` structure are as follows:

`token`

The value of the `token` item is the class token (§4.3.7.2) assigned to this class or interface.

`access_flags`

The value of the `access_flags` item is a mask of modifiers used with class and interface declarations. The `access_flags` modifiers are shown in the fol-

lowing table.

Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package	Class, interface
ACC_FINAL	0x0010	Is final; no subclasses allowed.	Class
ACC_INTERFACE	0x0200	Is an interface	Interface
ACC_ABSTRACT	0x0400	Is abstract; may not be instantiated	Class, interface
ACC_SHAREABLE	0x0800	Is shareable, may be shared between Java Card applets.	Class, interface

TABLE 5-3 Export file class access and modifier flags

The ACC_SHAREABLE flag indicates whether this class or interface is shareable.[1] A class is shareable if it implements (directly or indirectly) the `javacard.framework.shareable` interface. An interface is shareable if it is or implements (directly or indirectly) the `javacard.framework.Shareable` interface.

All other class access and modifier flags are defined in the same way and with the same restrictions as described in *The Java™ Virtual Machine Specification.*

The Java Card virtual machine reserves all other flag values. Their values must be zero.

`name_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§5.4.4) structure representing a valid Java class name stored as a simple (not fully qualified) name, that is, as a Java identifier.[2]

`export_interfaces_count`

The value of the `export_interface_count` item indicates the number of entries in the `interfaces` array.

1. The ACC_SHAREABLE flag is defined to enable Java Card virtual machines to implement the firewall restrictions defined by the *Java Card™ 2.1 Runtime Environment (JCRE) Specification.*

2. In Java class files class names are fully qualified. In Java Card `export` files all classes and interfaces enumerated are defined in the package of the `export` file making it unnecessary for class names to be fully qualified.

`interfaces[]`

The `interfaces` array contains an entry for each public interface implemented by this class or interface. It does not include package visible interfaces. It does include all public superinterfaces in the hierarchy of public interfaces implemented by this class or interface.

Each value in the `interfaces` array must be a valid index into the `constant_pool` table. The `constant_pool` entry at each value of `interfaces[i]`, where 0 <= i < `export_interfaces_count`, must be a `CONSTANT_Interfaceref_info` structure representing an interface which is an public superinterface of this class or interface type, in the left-to-right order given in the source for the type and its superclasses or superinterfaces.

`export_fields_count`

The value of the `export_fields_count` item gives the number of entries in the `fields` table.

`fields[]`

Each value in the `fields` table is a variable-length `field_info` (§5.6) structure. The `field_info` contains an entry for each publicly accessible field, both class variables and instance variables, declared by this class or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

`export_methods_count`

The value of the `export_methods_count` item gives the number of entries in the `methods` table.

`methods[]`

Each value in the `methods` table is a `method_info` (§5.7) structure. The `method_info` structure contains an entry for each publicly accessible class (static or constructor) method defined by this class, and each publicly accessible instance method defined by this class or its superclasses, or defined by this interface or its super-interfaces.

5.6 Fields

Each field is described by a variable-length `field_info` structure. The format of this structure is:

```
field_info {
       u1 token
       u2 access_flags
       u2 name_index
       u2 descriptor_index
       u2 attributes_count
       attribute_info attributes[attributes_count]
}
```

The items of the `field_info` structure are as follows:

`token`

The `token` item is the token assigned to this field. There are three scopes for field tokens: `final static` fields of primitive types (compile-time constants), all other `static` fields, and instance fields.

If this field is a compile-time constant, the value of the `token` item is 0xFFFF. Compile-time constants are represented in `export` files, but are not assigned token values suitable for late binding. Instead Java Card Converters must replace bytecodes that reference `final static` fields with bytecodes that load the constant value of the field.[1]

If this field is `static`, but is not a compile-time constant, the token item represents a static field token (§4.3.7.3).

If this field is an instance field, the token item represents an instance field token (§4.3.7.5).

1. Although Java compilers ordinarily replace references to final static fields of primitive types with primitive constants, this functionality is not required.

`access_flags`

The value of the `access_flags` item is a mask of modifiers used with fields. The `access_flags` modifiers are shown in the following table.

Name	Value	Meaning	Used By
`ACC_PUBLIC`	`0x0001`	Is public; may be accessed from outside its package.	Any field
`ACC_PROTECTED`	`0x0004`	Is protected; may be accessed within subclasses.	Class field
`ACC_STATIC`	`0x0008`	Is static.	Class field
`ACC_FINAL`	`0x0010`	Is final; no further overriding or assignment after initialization.	Any field

TABLE 5-4 Export file field access and modifier flags

Field access and modifier flags are defined in the same way and with the same restrictions as described in *The Java™ Virtual Machine Specification*.

The Java Card virtual machine reserves all other flag values. Their values must be zero.

`name_index`

The value of the `name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§5.4.4) structure representing a valid Java field name stored as a simple (not fully qualified) name, that is, as a Java identifier.

`descriptor_index`

The value of the `descriptor_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§5.4.4) structure representing a valid Java field descriptor.

Representation of a field descriptor in an `export` file is the same as in a Java `class` file. See the specification described in *The Java™ Virtual Machine Specification (*§ 4.3.2).

`attributes_count`

The value of the `attributes_count` item indicates the number of additional attributes of this field. The only `field_info` attribute currently defined is the `ConstantValue` attribute (§5.8.1). For `static final` fields of primitive types, the value must be 1; that is, when both the `ACC_STATIC` and `ACC_FINAL` bits in the flags item are set an attribute must be present. For all other fields the value

of the `attributes_count` item must be 0.

`attributes[]`

The only attribute defined for the `attributes` table of a `field_info` structure by this specification is the `ConstantValue` attribute (§5.8.1). This must be defined for `static final` fields of primitives (`boolean`, `byte`, `short`, and `int`).

5.7 Methods

Each method is described by a variable-length `method_info` structure. The format of this structure is:

```
method_info {
        u1 token
        u2 access_flags
        u2 name_index
        u2 descriptor_index
}
```

The items of the `method_info` structure are as follows:

`token`

The `token` item is the token assigned to this method. If this method is a static method or constructor, the `token` item represents a static method token (§4.3.7.4). If this method is a virtual method, the `token` item represents a virtual method token (§4.3.7.6).

access_flags

The value of the access_flags item is a mask of modifiers used with methods. The access_flags modifiers are shown in the following table.

Name	Value	Meaning	Used By
ACC_PUBLIC	0x0001	Is public; may be accessed from outside its package.	Any method
ACC_PROTECTED	0x0004	Is protected; may be accessed within subclasses.	Class/ instance method
ACC_STATIC	0x0008	Is static.	Class/ instance method
ACC_FINAL	0x0010	Is final; no further overriding or assignment after initialization.	Class/ instance method
ACC_ABSTRACT	0x0400	Is abstract; no implementation is provided	Any method

TABLE 5-5 Export file method access and modifier flags

Method access and modifier flags are defined in the same way and with the same restrictions as described in *The Java™ Virtual Machine Specification.*

Unlike in Java class files, the ACC_NATIVE flag is not supported in export files. Whether a method is native is an implementation detail that is not relevant to importing packages. The Java Card virtual machine reserves all other flag values. Their values must be zero.

name_index

The value of the name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§5.4.4) structure representing either the special internal method name for constructors, <init>, or a valid Java method name stored as a simple (not fully qualified) name.

descriptor_index

The value of the descriptor_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§5.4.4) structure representing a valid Java method descriptor.

Representation of a method descriptor in an export file is the same as in a Java class file. See the specification described in *The Java™ Virtual Machine Specification* (§ 4.3.3).

5.8 Attributes

Attributes are used in the `field_info` (§5.6) structure of the `export` file format. All attributes have the following general format:

```
attribute_info {
       u2 attribute_name_index
       u4 attribute_length
       u1 info[attribute_length]
}
```

5.8.1 ConstantValue Attribute

The `ConstantValue` attribute is a fixed-length attribute used in the `attributes` table of the `field_info` structures. A `ConstantValue` attribute represents the value of a `final static` field (compile-time constant); that is, both the `ACC_STATIC` and `ACC_FINAL` bits in the `flags` item of the `field_info` structure must be set. There can be no more than one `ConstantValue` attribute in the `attributes` table of a given `field_info` structure.

The `ConstantValue` attribute has the format:

```
ConstantValue_attribute {
       u2 attribute_name_index
       u4 attribute_length
       u2 constantvalue_index
}
```

The items of the `ConstantValue_attribute` structure are as follows:

`attribute_name_index`

The value of the `attribute_name_index` item must be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Utf8_info` (§5.4.4) structure representing the string "ConstantValue."

`attribute_length`

The value of the `attribute_length` item of a `ConstantValue_attribute` structure must be 2.

`constantvalue_index`

The value of the `constantvalue_index` item must be a valid index into the

`constant_pool` table. The `constant_pool` entry at that index must give the constant value represented by this attribute.

The `constant_pool` entry must be of a type `CONSTANT_Integer` (§5.4.3).

CHAPTER 6

The CAP File Format

This chapter describes the Java Card CAP (converted applet) file format. Each CAP file contains all of the classes and interfaces defined in one Java package. Java Card Converters must be capable of producing CAP files that conform to the specification provided in this chapter.

A CAP file consists of a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two and four consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high-order bytes come first. The first bit read of an 8-bit quantity is considered the *high bit*.

This chapter defines its own set of data types representing Java Card CAP file data: The types u1, and u2 represent an unsigned one-, and two-byte quantities, respectively. Some u1 types are represented as *bitfield* structures, consisting of arrays of bits. The zeroeth bit in each bit array represents the most significant bit, or *high bit*.

The Java Card CAP file format is presented using pseudo structures written in a C-like structure notation. To avoid confusion with the fields of Java Card virtual machine classes and class instances, the contents of the structures describing the Java Card CAP file format are referred to as *items*. Unlike the fields of a C structure, successive items are stored in the Java Card file sequentially, without padding or alignment.

Variable-sized *tables*, consisting of variable-sized items, are used in several CAP file data structures. Although we will use C-like array syntax to refer to table items, the fact that tables are streams of variable-sized structures means that it is not possible to directly translate a table index into a byte offset into the table.

A data structure referred to as an *array* consists of items equal in size.

Some items in the structures of the CAP file format are describe using a C-like *union* notation. The bytes contained in a union structure have one of the two formats. Selection of the two formats is based on the value of the high bit of the structure.

6.1 Component Model

A Java Card CAP file consists of a set of components. Each component describes a set of elements in the Java package defined, or an aspect of the CAP file. A complete CAP file must contain all of the required components specified in this chapter. Two components are optional: the Applet Component (§6.5) and Export Component (§6.12). The Applet Component is included only if one or more Applets are defined in the package. The Export Component is included only if classes in other packages may import elements in the package defined.

The content of each component defined in a CAP file must conform to the corresponding format specified in this chapter. All components have the following general format:

```
component {
       u1 tag
       u2 size
       u1 info[]
}
```

Each component begins with a 1-byte `tag` indicating the kind of component. Valid tags and their values are listed in TABLE 6-1. The `size` item indicates the number of bytes in the info array of the component, not including the `tag` and `size` items.

The content and format of the `info` array varies with the type of component.

Component Type	Value
COMPONENT_Header	1
COMPONENT_Directory	2
COMPONENT_Applet	3
COMPONENT_Import	4
COMPONENT_ConstantPool	5
COMPONENT_Class	6
COMPONENT_Method	7
COMPONENT_StaticField	8
COMPONENT_ReferenceLocation	9
COMPONENT_Export	10
COMPONENT_Descriptor	11

TABLE 6-1 CAP file component tags

Sun may define additional components in future versions of this Java Card vitural machine specification. It is guaranteed that additional components will have tag values between 12 and 127, inclusive.

6.1.1 Containment in a JAR File

All CAP file components are stored in individual files contained in a JAR File. The component file names are enumerated in TABLE 6-2. These names are not case sensitive.

Component Type	File Name
COMPONENT_Header	Header.cap
COMPONENT_Directory	Directory.cap
COMPONENT_Applet	Applet.cap
COMPONENT_Import	Import.cap
COMPONENT_ConstantPool	ConstantPool.cap
COMPONENT_Class	Class.cap
COMPONENT_Method	Method.cap
COMPONENT_StaticField	StaticField.cap
COMPONENT_ReferenceLocation	RefLocation.cap
COMPONENT_Export	Export.cap
COMPONENT_Descriptor	Descriptor.cap

TABLE 6-2 CAP file component file names

As described in §4.1.3, the path to the CAP file component files in a JAR file consists of a directory called `javacard` that is in a subdirectory representing the package's directory. For example, the CAP file component files of the package `javacard.framework` are located in the subdirectory `javacard/framework/javacard`.

The name of a JAR file containing CAP file component files is not defined as part of this specification. Other files, including other CAP files, may also reside in a JAR file that contains CAP file component files.

6.1.2 Defining New Components

Java Card CAP files are permitted to contain new, or custom, components. All new components not defined as part of this specification must not affect the semantics of the specified components, and Java Card virtual machines must be able to accept CAP files that do not contain new components. Java Card virtual machine implementations are required to silently ignore components they do not recognize.

New components are identified in two ways: they are assigned both an ISO 7816-5 AID (§4.2) and a tag value. Valid tag values are between 128 and 255, inclusive. Both of these identifiers are recorded in the `custom_component` item of the Directory Component (§6.4).

The new component must conform to the general component format defined in this chapter, with a `tag` value, a `size` value indicating the number of bytes in the component (excluding the `tag` and `size` items), and an `info` item containing the content of the new component.

A new component file is stored in a JAR file, following the same restrictions as those specified in §4.1.3. That is, the file containing the new component must be located in the `<package_directory>/javacard` subdirectory of the JAR file and must have the extension '`.cap`'.

6.2 Installation

Installing a `CAP` file onto a Java Card enabled device entails communication between a Java Card enabled terminal and that device. While it is beyond the scope of this specification to define an installation protocol between a terminal and a device, the `CAP` file component order shown in TABLE 6-3 is a reference load order suitable for an implementation with a simple memory management model on a limited memory device.

Component Type
COMPONENT_Header
COMPONENT_Directory
COMPONENT_Import
COMPONENT_Applet
COMPONENT_Class
COMPONENT_Method
COMPONENT_StaticField
COMPONENT_Export
COMPONENT_ConstantPool
COMPONENT_ReferenceLocation
COMPONENT_Descriptor (optional)

TABLE 6-3 Reference component install order

6.3 Header Component

The Header Component contains general information about this CAP file and the package it defines. It is described by the following variable-length structure:

```
header_component {
       u1 tag
       u2 size
       u4 magic
       u1 minor_version
       u1 major_version
       u1 flags
       package_info this_package
}
```

The items in the header_component structure are as follows:

tag

The tag item has the value COMPONENT_Header (1).

size

The size item indicates the number of bytes in the header_component structure, excluding the tag and size items. The value of the size item must be greater than zero.

magic

The magic item supplies the magic number identifying the Java Card CAP file format; it has the value 0xDECAFFED.

minor_version, major_version

The minor_version and major_version items are the minor and major version numbers of this CAP file. An implementation of a Java Card virtual machine must support CAP files having a specific major version number and minor version numbers in the range of 0 through some particular minor_version.

If a Java Card virtual machine encounters a CAP file with the supported major version but an unsupported minor version, the Java Card virtual machine must not attempt to interpret the content of the CAP file. However, it will be feasible to upgrade a Java Card virtual machine to support the newer minor version.

A Java Card virtual machine must not attempt to interpret a CAP file with a different major version. A change of the major version number indicates a major incompatibility change, one that requires a fundamentally different Java Card

virtual machine.

In this specification, the major version of the CAP file has the value 2 and the minor version has the value 1. Only Sun Microsystems, Inc. may define the meaning and values of new CAP file versions.

flags

The flags item is a mask of modifiers that apply to this package. The flags modifiers are shown in the following table.

Flags	Value
ACC_INT	0x01
ACC_EXPORT	0x02
ACC_APPLET	0x04

TABLE 6-4 CAP file package flags

The ACC_INT flag has the value of one if the Java int type is used in this package. The int type is used if one or more of the following is present:

- a parameter to a method of type int,
- a local variable of type int,
- a field of type int,
- a field of type int array, or
- an instruction of type int.

Otherwise the ACC_INT flag has the value of 0.

The ACC_EXPORT flag has the value of one if an Export Component (§6.12) is included in this CAP file. Otherwise it has the value of 0.

The ACC_APPLET flag has the value of one if an Applet Component (§6.5) is included in this CAP file. Otherwise it has the value of 0.

All other bits in the flags item not defined in TABLE 6-4 are reserved for future use. Their values must be zero and they must be ignored by Java Card virtual machines.

this_package

The this_package item describes the package defined in this CAP file. It is represented as a package_info structure:

```
package_info {
       u1 minor_version
       u1 major_version
       u1 AID_length
       u1 AID[AID_length]
}
```

The items in the package_info structure are as follows:

minor_version, major_version

The minor_version and major_version items are the minor and major version numbers of this package. These values uniquely identify the particular implementation of this package and indicate the binary compatibility between packages. See §4.5 for a description of assigining and using package version numbers.

AID_length

The AID_length item represents the number of bytes in the AID item. Valid values are between 5 and 16, inclusive.

AID[]

The AID item represents the Java Card name of the package. See ISO 7816-5 for the definition of an AID (§4.2).

6.4 Directory Component

The Directory Component lists the size of each of the components defined in this CAP file. When an optional component is not included, such as the Applet Component (§6.5) or Export Component (§6.12), it is represented in the Directory Component with size equal to zero. The Directory Component also includes entries for new (or custom) components.

The Directory Component is described by the following variable-length structure:

```
directory_component {
      u1 tag
      u2 size
      u2 component_sizes[11]
      static_field_size_info static_field_size
      u1 import_count
      u1 applet_count
      u1 custom_count
      custom_component_info custom_components[custom_count]
}
```

The items in the directory_component structure are as follows:

tag

The tag item has the value COMPONENT_Directory (2).

`size`

The `size` item indicates the number of bytes in the `directory_component` structure, excluding the `tag` and `size` items. The value of the `size` item must be greater than zero.

`component_sizes[]`

The `component_sizes` item is an array representing the number of bytes in each of the components in this `CAP` file. All of the 11 components defined in this chapter are represented in the `component_sizes` array. The value of an index into the array is equal to the value of the tag of the component represented at that entry, minus 1.

The value in each entry in the `component_sizes` array is that same as the `size` item in the corresponding component. It represents the number of bytes in the component, excluding the `tag` and `size` items.

The value of an entry in the `component_sizes` array is zero for components not included in this `CAP` file. Components that may not be included are the Applet Component (§6.5) and the Export Component (§6.12). For all other components the value is greater than zero.

`static_field_size`

The `static_field_size` item is a `static_field_size_info` structure. The structure is defined as:

```
static_field_size_info {
       u2 image_size
       u2 array_init_count
       u2 array_init_size
}
```

The items in the `static_field_size_info` structure are the following:

`image_size`

The `image_size` item has the same value as the `image_size` item in the Static Field Component (§6.10). It represents the total number of bytes in the `static` fields defined in this package, excluding `final static` fields of primitive types.

`array_init_count`

The `array_init_count` item has the same value as the `array_init_count` item in the Static Field Component (§6.10). It represents the number of arrays initialized in all of the `<clinit>` methods in this package.

`array_init_size`

The `array_init_size` item represents the sum of the `count` items in

the array_init table item of the Static Field Component (§6.10). It is the total number of bytes in all of the arrays initialized in all of the <clinit> methods in this package.

import_count

The import_count item indicates the number of packages imported by classes and interfaces in this package. This item has the same value as the count item in the Import Component (§6.6).

applet_count

The applet_count item indicates the number of applets defined in this package. If an Applet Component (§6.5) is not included in this CAP file, the value of the applet_count item is zero. Otherwise the value of the applet_count item is the same as the value of the count item in the Applet Component (§6.5).

custom_count

The custom_count item indicates the number of entries in the custom_components table. Valid values are between 0 and 127, inclusive.

custom_components[]

The custom_components item is a table of variable-length custom_component_info structures. Each new component defined in this CAP file must be represented in the table. These components are not defined in this standard.

The custom_component_info structure is defined as:

```
custom_component_info {
       u1 component_tag
       u1 size
       u1 AID_length
       u1 AID[AID_length]
}
```

The items in entries of the custom_component_info structure are:

component_tag

The component_tag item represents the tag of the component. Valid values are between 128 and 255, inclusive.

size

The size item represents the number of bytes in the component, excluding the tag and size items.

AID_length

The AID_length item represents the number of bytes in the AID item.

Valid values are between 5 and 16, inclusive.

`AID[]`

The `AID` item represents the Java Card name of the component. See ISO 7816-5 for the definition of an AID (§4.2).

Each component is assigned an AID conforming to the ISO 7816-5 standard. The RID (first 5 bytes) of all of the custom component AIDs must have the same value. In addition, the RID of the custom component AIDs must have the same value as the RID of the package defined in this `CAP` file.

6.5 Applet Component

The Applet Component contains an entry for each of the applets defined in this package. Applets are defined by implementing a non-abstract subclass, direct or indirect, of the `javacard.framework.Applet` class.[1] If no applets are defined, this component must not be present in this `CAP` file.

The Applet Component is described by the following variable-length structure:

```
applet_component {
       u1 tag
       u2 size
       u1 count
       { u1 AID_length
         u1 AID[AID_length]
         u2 install_method_offset
       } applets[count]
}
```

The items in the `applet_component` structure are as follows:

`tag`

The `tag` item has the value `COMPONENT_Applet` (3).

`size`

The `size` item indicates the number of bytes in the `applet_component` structure, excluding the `tag` and `size` items. The value of the `size` item must be greater than zero.

1. Restrictions placed on an applet definition are imposed by the Java Card Runtime Environment (JCRE) 2.1 specification.

`count`

The `count` item indicates the number of applets defined in this package.

`applets[]`

The `applets` item represents a table of variable-length structures each describing an applet defined in this package.

The items in each entry of the `applets` table are defined as follows:

`AID_length`

The `AID_length` item represents the number of bytes in the `AID` item. Valid values are between 5 and 16, inclusive.

`AID[]`

The `AID` item represents the Java Card name of the applet.

Each applet is assigned an AID conforming to the ISO 7816-5 standard (§4.2). The RID (first 5 bytes) of all of the applet AIDs must have the same value. In addition, the RID of each applet AIDs must have the same value as the RID of the package defined in this `CAP` file.

`install_method_offset`

The value of the `install_method_offset` item must be a 16-bit offset into the `info` item of the Method Component (§6.9). The item at that offset must be a `method_info` structure that represents the static `install(byte[],short,byte)` method of the applet.[1] The `install(byte[],short,byte)` method must be defined in a class that extends the `javacard.framework.applet` class, directly or indirectly. The `install(byte[],short,byte)` method is called to initialize the applet.

1. Restrictions placed on the `install(byte[],short,byte)` method of an applet are imposed by the Java Card Runtime Environment (JCRE) 2.1 specification.

6.6 Import Component

The Import Component lists the set of packages imported by the classes in this package. It does not include an entry for the package defined in this CAP file. The Import Component is represented by the following structure:

```
import_component {
       u1 tag
       u2 size
       u1 count
       package_info packages[count]
}
```

The items in the import_component structure are as follows:

tag

The tag item has the value COMPONENT_Import (4).

size

The size item indicates the number of bytes in the import_component structure, excluding the tag and size items. The value of the size item must be greater than zero.

count

The count item indicates the number of items in the packages table. The value of the count item must be between 0 and 127, inclusive.

packages[]

The packages item represents a table of variable-length package_info structures as defined for this_package under §6.3. The table contains an entry for each of the packages referenced in the CAP file, not including the package defined.

The major and minor version numbers specified in the package_info structure are equal to the major and minor versions specified in the imported package's export file. See §4.5 for a description of assigining and using package version numbers.

Components of this CAP file refer to an imported package by using a index in this packages table. The index is called a *package token* (§4.3.7.1).

6.7 Constant Pool Component

The Constant Pool Component contains an entry for each of the classes, methods, and fields referenced by elements in the Method Component (§6.9) of this CAP file. The referencing elements in the Method Component may be instructions in the methods or exception handler catch types in the exception handler table.

Entries in the Constant Pool Component reference elements in the Class Component (§6.8), Method Component (§6.9), and Static Field Component (§6.10). The Import Component (§6.6) is also accessed using a package token (§4.3.7.1) to describe references to classes, methods and fields defined in imported packages. Entries in the Constant Pool Component do not reference other entries internal to itself.

The Constant Pool Component is described by the following structure:

```
constant_pool_component {
        u1 tag
        u2 size
        u2 count
        cp_info constant_pool[count]
}
```

The items in the `constant_pool_component` structure are as follows:

`tag`

The `tag` item has the value `COMPONENT_ConstantPool` (5).

`size`

The `size` item indicates the number of bytes in the `constant_pool_component` structure, excluding the `tag` and `size` items. The value of the `size` item must be greater than zero.

`count`

The `count` item represents the number entries in the `constant_pool` array. Valid values are between 0 and 65535, inclusive.

`constant_pool`

The `constant_pool` item represents an array of `cp_info` structures:

```
cp_info {
        u1 tag
        u1 info[3]
}
```

Each item in the `constant_pool` array is a 4-byte structure. Each structure

must begin with a 1-byte `tag` indicating the kind of `cp_info` entry. The content and format of the 3-byte `info` array varies with the value of the tag. The valid tags and their values are listed in the following table.

Constant Type	Tag
`CONSTANT_Classref`	1
`CONSTANT_InstanceFieldref`	2
`CONSTANT_VirtualMethodref`	3
`CONSTANT_SuperMethodref`	4
`CONSTANT_StaticFieldref`	5
`CONSTANT_StaticMethodref`	6

TABLE 6-5 CAP file constant pool tags

Java Card constant types are more specific than those in Java `class` files. The categories indicate not only the type of the item referenced, but also the manner in which it is referenced.

For example, in the Java constant pool there is one constant type for method references, while in the Java Card constant pool there are three constant types for method references: one for virtual method invocations using the *invokevirtual* bytecode, one for super method invocations using the *invokespecial* bytecode, and one for static method invocations using either the *invokestatic* or *invokespecial* bytecode.[1] The additional information provided by a constant type in Java Card technologies simplifies resolution of references.

There are no ordering constraints on constant pool entries. It is recommended, however, that `CONSTANT_InstanceFieldref` (§6.7.2) constants occur early in the array to permit using *getfield_T* and *putfield_T* bytecodes instead of *getfield_T_w* and *putfield_T_w* bytecodes. The former have 1-byte constant pool index parameters while the latter have 2-byte constant pool index parameters.

1. The constant pool index parameter of an *invokespecial* bytecode is to a `CONSTANT_StaticMethodref` when the method referenced is a constructor or a private virtual method. In these cases the method invoked is fully known when the CAP file is created. In the cases of virtual method and super method references, the method invoked is dependent upon an instance of a class and its hierarchy, both of which may be partially unknown when the CAP file is created.

6.7.1 CONSTANT_Classref

The `CONSTANT_Classref_info` structure is used to represent a reference to a class or an interface. The class or interface may be defined in this package or in an imported package.

```
CONSTANT_Classref_info {
       u1 tag
       union {
              u2 internal_class_ref
              { u1 package_token
                u1 class_token
              } external_class_ref
       } class_ref
       u1 padding
}
```

The items in the `CONSTANT_Classref_info` structure are the following:

`tag`

The `tag` item has the value `CONSTANT_Classref` (1).

`class_ref`

The `class_ref` item represents a reference to a class or interface. If the class or interface is defined in this package the structure represents an `internal_class_ref` and the high bit of the structure is zero. If the class or interface is defined in another package the structure represents an `external_class_ref` and the high bit of the structure is one.

`internal_class_ref`

The `internal_class_ref` structure represents a 16-bit offset into the `info` item of the Class Component (§6.8) to an `interface_info` or `class_info` structure. The `interface_info` or `class_info` structure must represent the referenced class or interface.

The value of the `internal_class_ref` item must between 0 and 32767, inclusive, making the high bit equal to zero.

`external_class_ref`

The `external_class_ref` structure represents a reference to a class or interface defined in an imported package. The high bit of this structure is one.

`package_token`

The `package_token` item represents a package token (§4.3.7.1) defined in the Import Component (§6.6) of this `CAP` file. The value of this token must be a valid index into the

`packages` table item of the `import_component` structure. The package represented at that index must be the imported package.

The value of the package token must be between 0 and 127, inclusive.

The high bit of the `package_token` item is equal to one.

`class_token`

The `class_token` item represents the token of the class or interface (§4.3.7.2) of the referenced class or interface. It has the value of the class token of the class as defined in the `Export` file of the imported package.

`padding`

The `padding` item has the value zero. It is present to make the size of a `CONSTANT_Classref_info` structure the same as all other constants in the `constant_pool` array.

6.7.2 CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, and CONSTANT_SuperMethodref

References to instance fields, and virtual methods are represented by similar structures:

```
CONSTANT_InstanceFieldref_info {
       u1 tag
       class_ref class
       u1 token
}

CONSTANT_VirtualMethodref_info {
       u1 tag
       class_ref class
       u1 token
}

CONSTANT_SuperMethodref_info {
       u1 tag
       class_ref class
       u1 token
}
```

The items in these structures are as follows:

`tag`

The tag item of a `CONSTANT_InstanceFieldref_info` structure has the value `CONSTANT_InstanceFieldref` (2).

The tag item of a `CONSTANT_VirtualMethodref_info` structure has the value `CONSTANT_VirtualMethodref` (3).

The tag item of a `CONSTANT_SuperMethodref_info` structure has the value `CONSTANT_SuperMethodref` (4).

`class`

The `class` item represents the class associated with the referenced instance field, virtual method, or `super` method invocation. It is a `class_ref` structure (§6.7.1). If the referenced class is defined in this package the high bit is equal to zero. If the reference class is defined in an imported package the high bit of this structure is equal to one.

The class referenced in the `CONSTANT_InstanceField_info` structure must be the class that contains the declaration of the instance field.

The class referenced in the `CONSTANT_VirtualMethodref_info` structure must be a class that contains a declaration or definition of the virtual method.

The class referenced in the `CONSTANT_SuperMethodref_info` structure must always be internal to the class that defines the method that contains the Java language-level `super` invocation. The class must be defined in this package.

`token`

The `token` item in the `CONSTANT_InstanceFieldref_info` structure represents an instance field token (§4.3.7.5) of the referenced field. The value of the instance field token is defined within the scope of the class indicated by the `class` item.

The `token` item of the `CONSTANT_VirtualMethodref_info` structure represents the virtual method token (§4.3.7.6) of the referenced method. The virtual method token is defined within the scope of the hierarchy of the class indicated by the `class` item. If the referenced method is `public` or `protected` the high bit of the `token` item is zero. If the referenced method is package-visible the high bit of the `token` item is one. In this case the `class` item must represent a reference to a class defined in this package.

The `token` item of the `CONSTANT_SuperMethodref_info` structure represents the virtual method token (§4.3.7.6) of the referenced method. Unlike in the CONSTANT_VirtualMethodref_info structure, the virtual method token is defined within the scope of the hierarchy of the superclass of the class indicated by the `class` item. If the referenced method is `public` or `protected` the

high bit of the `token` item is zero. If the referenced method is package-visible the high bit of the `token` item is one. In the latter case the `class` item must represent a reference to a class defined in this package and at least one super-class of the class that contains a defintion of the virtual method must also be defined in this package.

6.7.3 CONSTANT_StaticFieldref and CONSTANT_StaticMethodref

References to static fields and methods are represented by similar structures:

```
CONSTANT_StaticFieldref_info {
       u1 tag
       union {
              { u1 padding
                u2 offset
              } internal_ref
              { u1 package_token
                u1 class_token
                u1 token
              } external_ref
       } static_field_ref
}

CONSTANT_StaticMethodref_info {
       u1 tag
       union {
              { u1 padding
                u2 offset
              } internal_ref
              { u1 package_token
                u1 class_token
                u1 token
              } external_ref
       } static_method_ref
}
```

The items in these structures are as follows:

`tag`

The `tag` item of a CONSTANT_StaticFieldref_info structure has the value CONSTANT_StaticFieldref (5).

The `tag` item of a CONSTANT_StaticMethodref_info structure has the value CONSTANT_StaticMethodref (6).

static_field_ref and static_method_ref

The static_field_ref and static_method_ref item represents a reference to a static field or static method, respectively. Static method references include references to static methods, constructors, and private virtual methods.

If the referenced item is defined in this package the structure represents an internal_ref and the high bit of the structure is zero. If the referenced item is defined in another package the structure represents an external_ref and the high bit of the structure is one.

internal_ref

The internal_ref item represents a reference to a static field or method defined in this package. The items in the structure are:

padding

The padding item is equal to 0.

offset

The offset item of a CONSTANT_StaticFieldref_info structure represents a 16-bit offset into the Static Field Image defined by the Static Field component (§6.10) to this static field.

The offset item of a CONSTANT_StaticMethodref_info structure represents a 16-bit offset into the info item of the Method Component (§6.9) to a method_info structure. The method_info structure must represent the referenced method.

external_ref

The external_ref item represents a reference to a static field or method defined in an imported package. The items in the structure are:

package_token

The package_token item represents a package token (§4.3.7.1) defined in the Import Component (§6.6) of this CAP file. The value of this token must be a valid index into the packages table item of the import_component structure. The package represented at that index must be the imported package.

The value of the package token must be between 0 and 127, inclusive.

The high bit of the package_token item is equal to one.

`class_token`

The `class_token` item represents the token (§4.3.7.2) of the class of the referenced class. It has the value of the class token of the class as defined in the `Export` file of the imported package.

The class indicated by the `class_token` item must define the referenced field or method.

`token`

The `token` item of a `CONSTANT_StaticFieldref_info` structure represents a static field token (§4.3.7.3) as defined in the `Export` file of the imported package. It has the value of the token of the referenced field.

The `token` item of a `CONSTANT_StaticMethodref_info` structure represents a static method token (§4.3.7.4) as defined in the `Export` file of the imported package. It has the value of the token of the referenced method.

6.8 Class Component

The Class Component describes each of the classes and interfaces defined in this package. It does not contain complete access information and content details for each class and interface. Instead, the information included is limited to that required to execute operations associated with a particular class or interface, without performing verification. Complete details regarding the classes and interfaces defined in this package are included in the Descriptor Component (§6.13).

The information included in the Class Component for each interface is sufficient to uniquely identify the interface and to test whether or not a cast to that interface is valid.

The information included in the Class Component for each class is sufficient to resolve operations associated with instances of a class. The operations include creating an instance, testing whether or not a cast of the instance is valid, dispatching virtual method invocations, and dispatching interface method invocations. Also included is sufficient information to locate instance fields of type `reference`, including arrays.

The classes represented in the Class Component reference other entries in the Class Component in the form of superclass, superinterface and implemented interface references. When a superclass, superinterface or implemented interface is defined in an imported package the Import Component is used in the representation of the reference.

The classes represented in the Class Component also contain references to virtual methods defined in the Method Component (§6.9) of this CAP file. References to virtual methods defined in imported packages are not explicitly described. Instead such methods are located through a superclass within the hierarchy of the class, where the superclass is defined in the same imported package as the virtual method.

The Constant Pool Component (§6.7), Export Component (§6.12) and Descriptor Component (§6.13) reference classes and interfaces defined in the Class Component. No other CAP file components reference the Class Component.

The Class Component is represented by the following structure:

```
class_component {
       u1 tag
       u2 size
       interface_info interfaces[]
       class_info classes[]
}
```

The items in the class_component structure are as follows:

tag

The tag item has the value COMPONENT_Class (6).

size

The size item indicates the number of bytes in the class_component structure, excluding the tag and size items. The value of the size item must be greater than zero.

interfaces[]

The interfaces item represents an array of interface_info structures. Each interface defined in this package is represented in the array. The entries are ordered based on hierarchy such that a superinterface has a lower index than any of its subinterfaces.

classes[]

The classes item represents a table of variable-length class_info structures. Each class defined in this package is represented in the array. The entries are ordered based on hierarchy such that a superclass has a lower index than any of its subclasses.

6.8.1 interface_info and class_info

The `interface_info` and `class_info` structures represent interfaces and classes, respectively. The two are differentiated by the value of the high bit in the structures. They are defined as follows:

```
interface_info {
  u1 bitfield {
    bit[4] flags
    bit[4] interface_count
  }
  class_ref superinterfaces[interface_count]
}

class_info {
  u1 bitfield {
    bit[4] flags
    bit[4] interface_count
  }
  class_ref super_class_ref
  u1 declared_instance_size
  u1 first_reference_index
  u1 reference_count
  u1 public_method_table_base
  u1 public_method_table_count
  u1 package_method_table_base
  u1 package_method_table_count
  u2 public_virtual_method_table[public_method_table_count]
  u2 package_virtual_method_table[package_method_table_count]
  implemented_interface_info interfaces[interface_count]
}
```

The items of the `interface_info` and `class_info` structure are as follows:

`flags`

The flags item is a mask of modifiers used to describe this interface or class. Valid values are shown in the following table:

Name	Value
ACC_INTERFACE	0x8
ACC_SHAREABLE	0x4

TABLE 6-6 CAP file interface and class flags

The ACC_INTERFACE flag indicates whether this `interface_info` or `class_info` structure represents an interface or a class. The value must be 1 if it represents an `interface_info` structure and 0 if a `class_info` structure.

The ACC_SHAREABLE flag in an `interface_info` structure indicates whether

this interface is shareable. The value of this flag must be one if and only if the interface is `javacard.framework.Shareable` interface or implements that interface directly or indirectly.

The `ACC_SHAREABLE` flag in a `class_info` structure indicates whether this class is shareable.[1] The value of this flag must be one if and only if this class or any of its superclasses implements an interface that is shareable.

The Java Card virtual machine reserves all other flag values. Their values must be zero.

`interface_count`

The `interface_count` item of the `interface_info` structure indicates the number of entries in the `superinterfaces` table item. The value represents the number of immediate superinterfaces of this interface.It does not include superinterfaces of the superinterfaces. Valid values are between 0 and 15, inclusive.

The `interface_count` item of the `class_info` structure indicates the number of entries in the `interfaces` table item. The value represents the number of interfaces implemented by this class, including superinterfaces of those interfaces and potentially interfaces implemented by superclasses of this class. Valid values are between 0 and 15, inclusive.

`superinterfaces`

The `superinterfaces` item of the `interface_info` structure is an array of `class_ref` structures representing the superinterfaces of this interface. The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (§6.7.1). This array is empty if this interface has no superinterfaces. Only immediate superinterfaces are represented in the array. Superinterfaces of superinterfaces are not included, and class `Object` is not included either.

`super_class_ref`

The `super_class_ref` item of the `class_info` structure is a `class_ref` structure representing the superclass of this class. The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (§6.7.1).

The `super_class_ref` item has the value of 0xFFFF only if this class does not have a superclass. Otherwise the value of the `super_class_ref` item is limited only by the constraints of the `class_ref` structure.

`declared_instance_size`

The `declared_instance_size` item of the `class_info` structure represents the number of 16-bit cells required to represent the instance fields declared by

1. A Java Card virtual machine uses the `ACC_SHAREABLE` flag to implement the firewall restrictions defined by the Java Card Runtime Environment (JCRE) 2.1 specification.

this class. It does not include instance fields declared by superclasses of this class.

Instance fields of type `int` are represented in two 16-bit cells, while all other field types are represented in one 16-bit cell.

`first_reference_token`

The `first_reference_token` item of the `class_info` structure represents the instance field token (§4.3.7.5) value of the first `reference` type instance field defined by this class. It does not include instance fields defined by superclasses of this class.

If this class does not define any `reference` type instance fields, the value of the `first_reference_token` is 0xFF. Otherwise the value of the `first_reference_token` item must be within the range of the set of instance field tokens of this class.

`reference_count`

The `reference_count` item of the `class_info` structure represents the number of `reference` type instance field defined by this class. It does not include `reference` type instance fields defined by superclasses of this class.

Valid values of the `reference_count` item are between 0 and the maximum number of instance fields defined by this class.

`public_method_table_base`

The `public_method_table_base` item of the `class_info` structure is equal to the virtual method token value (§4.3.7.6) of the first method in the `public_virtual_method_table` array. If the `public_virtual_method_table` array is empty the value of the `public_method_table_base` item is equal to the `public_method_table_base` item of the `class_info` structure of this class' superclass plus the `public_method_table_count` item of the `class_info` structure of this class' superclass. If this class has no superclass and the `public_virtual_method_table` array is empty, the value of the `public_method_table_base` item is zero.

`public_method_table_count`

The `public_method_table_count` item of the `class_info` structure indicates the number of entries in the `public_virtual_method_table` array.

If this class does not define any `public` or `protected` override methods, the minimum valid value of `public_method_table_count` item is the number of `public` and `protected` virtual methods declared by this class. If this class defines one or more `public` or `protected` override methods, the minimum valid value of `public_method_table_count` item is the value of the largest `public` or `protected` virtual method token, minus the value of the smallest

public or protected virtual override method token, plus one.

The maximum valid value of the public_method_table_count item is the value of the largest public or protected virtual method token, plus one.

Any value for the public_method_table_count item between the minimum and maximum specified here is valid. However, the value must correspond to the number of entries in the public_virtual_method_table array.

package_method_table_base

The package_method_table_base item of the class_info structure is equal to the virtual method token value (§4.3.7.6) of the first entry in the package_virtual_method_table array. If the package_virtual_method_table array is empty the value of the package_method_table_base item is equal to the package_method_table_base item of the class_info structure of this class' superclass plus the package_method_table_count item of the class_info structure of this class' superclass. If this class has no superclass or inherits from a class defined in another package and the package_virtual_method_table array is empty, the value of the package_method_table_base item is zero.

package_method_table_count

The package_method_table_count item of the class_info structure indicates the number of entries in the package_virtual_method_table array.

If this class does not define any override methods, the minimum valid value of package_method_table_count item is the number of package visible virtual methods declared by this class. If this class defines one or more package visible override methods, the minimum valid value of package_method_table_count item is the value of the largest package visible virtual method token, minus the value of the smallest package visible virtual override method token, plus one.

The maximum valid value of the package_method_table_count item is the value of the largest package visible method token, plus one.

Any value for the package_method_table_count item between the minimum and maximum specified here are valid. However, the value must correspond to the number of entries in the package_virtual_method_table.

public_virtual_method_table

The public_virtual_method_table item of the class_info structure represents an array of public and protected virtual methods. These methods can be invoked on an instance of this class. The public_virtual_method_table array includes methods declared or defined by this class. It may also include methods declared or defined by any or all of its superclasses. The value of an index into this table must be equal to the value of the virtual method token of

the indicated method, minus the value of the `public_method_table_base` item.

Entries in the `public_virtual_method_table` array that represent methods defined or declared in this package contain offsets into the `info` item of the Method Component (§6.9) to the `method_info` structure representing the method. Entries that represent methods defined or declared in an imported package contain the value 0xFFFF.

Entries for methods that are declared abstract, not including those defined by interfaces, are represented in the `public_virtual_method_table` array in the same way as non-abstract methods.

`package_virtual_method_table`

The `package_virtual_method_table` item of the `class_info` structure represents an array of package-visible virtual methods. These methods can be invoked on an instance of this class. The `package_virtual_method_table` array includes methods declared or defined by this class. It may also include methods declared or defined by any or all of its superclasses that are defined in this package. The value of an index into this table must be equal to the value of the virtual method token of the indicated method & 0x7F, minus the value of the `package_method_table_base` item.

All entries in the `package_virtual_method_table` array represent methods defined or declared in this package. They contain offsets into the `info` item of the Method Component (§6.9) to the `method_info` structure representing the method.

Entries for methods that are declared abstract, not including those defined by interfaces, are represented in the `package_virtual_method_table` array in the same way as non-abstract methods.

`interfaces[]`

The `interfaces` item of the `class_info` structure represents a table of variable-length `implemented_interface_info` structures. The table must contain an entry for each of the implemented interfaces indicated in the declaration of this class and each of the interfaces in the hierarchies of those interfaces. Interfaces that occur more than once are represented by a single entry. Interfaces implemented by superclasses of this class may optionally be represented.

Given the declarations below, the number of entries for class c0 is 1 and the entry in the `interfaces` array is i0. The minimum number of entries for class c1 is 3 and the entries in the `interfaces` array are i1, i2, and i3. The entries for class c1 may also include interface i0, which is implemented by the superclass of c1.

```
interface i0 {}
interface i1 {}
interface i2 extends i1 {}
interface i3 {}
class c0 implements i0 {}
class c1 extends c0 implements i2, i3 {}
```

The `implemented_interface_info` structure is defined as follows:

```
implemented_interface_info {
       class_ref interface
       u1 count
       u1 index[count]
}
```

The items in the `implemented_interface_info` structure are defined as follows:

`interface`

The `interface` item has the form of a `class_ref` structure. The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (§6.7.1). The `interface_info` structure referenced by the `interface` item represents an interface implemented by this class.

`count`

The `count` item indicates the number of entries in the `index` array.

`index`

The `index` item is an array that maps declarations of interface methods to implementations of those methods in this class. It is a representation of a the set of methods declared by the interface and its superinterfaces.

Entries in the `index` array must be ordered such that the interface method token value (§4.3.7.7) of the interface method is equal to the index into the array. The interface method token value is assigned to the method within the scope of the interface definition and its superinterfaces, not within the scope of this class.

The values in the `index` array represent the virtual method tokens (§4.3.7.6) of the implementations of the interface methods. The virtual method token values are defined within the scope of the hierarchy of this class.

6.9 Method Component

The Method Component describes each of the methods declared in this package, excluding `<clinit>` methods and interface method declarations. The exception handlers associated with each method are also described.

The Method Component does not contain complete access information and descriptive details for each method. Instead, the information is limited to that required to execute each method, without performing verification. Complete details regarding the methods defined in this package are included in the Descriptor Component (§6.13).

Instructions and exception handler catch types in the Method Component reference entries in the Constant Pool Component (§6.7). No other `CAP` file components, including the Method Component, are referenced by the elements in the Method Component.

The Applet Component (§6.5), Constant Pool Component (§6.7), Export Component (§6.12), and Descriptor Component (§6.13) reference methods defined in the Method Component. The Reference Location Component (§6.11) references all constant pool indices contained in the Method Component. No other `CAP` file components reference the Method Component.

The Method Component is represented by the following structure:

```
method_component {
       u1 tag
       u2 size
       u1 handler_count
       exception_handler_info
       exception_handlers[handler_count]
       method_info methods[]
}
```

The items in the `method_component` structure are as follows:

`tag`

The `tag` item has the value `COMPONENT_Method` (7).

`size`

The `size` item indicates the number of bytes in the `method_component` structure, excluding the `tag` and `size` items. The value of the `size` item must be greater than zero.

`handler_count`

The `handler_count` item represents the number of entries in the `exception_handlers` array. Valid values are between 0 and 255, inclusive.

`exception_handlers[]`

The `exception_handlers` item represents an array of 8-byte `exception_handler_info` structures. Each `exception_handler_info` structure represents a `catch` or `finally` block defined in a method of this package.

Entries in the `exception_handlers` array are sorted in ascending order by the distance between the beginning of the Method Component to the endpoint of each exception handler range in the `methods` item.

`methods[]`

The `methods` item represents a table of variable-length `method_info` structures. Each entry represents a method declared in a class of this package. `<clinit>` methods and interface method declaration are not included; all other methods, including non-interface abstract methods, are.

6.9.1 exception_handler_info

The `exception_handler_info` structure is defined as follows:

```
exception_handler_info {
       u2 start_offset
       u2 active_length
       u2 handler_offset
       u2 catch_type_index
}
```

The items in the `exception_handler_info` structure are as follows:

`start_offset, active_length`

The `active_length` item is encoded to indicate whether the active range of this exception handler is nested within another exception handler. The high bit of the `active_length` item is equal to 1 if the active range is not contained within another exception handler, and this exception handler is the last handler applicable to the active range. The high bit is equal to 0 if the active range is contained within the active range of another exception handler, or there are successive handlers applicable to the same active range.

end_offset is defined as `start_offset` plus `active_length` & 0x7FFF.

The `start_offset` item and *end_offset* are byte offsets into the `info` item of

the Method Component. They indicate the ranges in a bytecode array at which the exception handler is active. The value of the `start_offset` must be a valid offset into a `bytecodes` array of a `method_info` structure to an opcode of an instruction. The value of the *end_offset* either must be a valid offset into a `bytecodes` array of a `method_info` structure to an opcode of an instruction or must be equal to a method's bytecode count, the length of the `bytecodes` array of a `method_info` structure. The value of the `start_offset` must be less than the value of the *end_offset*.

The `start_offset` is inclusive and the *end_offset* is exclusive; that is, the exception handler must be active while the execution address is within the interval [`start_offset`, *end_offset*).

`handler_offset`

The handler_offset item represents a byte offset into the `info` item of the Method Component. It indicates the start of the exception handler. The value of the item must be a valid offset into a `bytecodes` array of a `method_info` structure to an opcode of an instruction, and must be less than the value of the method's bytecode count.

`catch_type_index`

If the value of the `catch_type_index` item is non-zero, it must be a valid index into the `constant_pool` array of the Constant Pool Component (§6.7). The `constant_pool` entry at that index must be a `CONSTANT_Classref_info` structure, representing the class of the exception caught by this `exception_handlers` array entry.

If the `exception_handlers` table entry represents a finally block, the value of the `catch_type_index` item is zero. In this case the exception handler is called for all exceptions that are thrown within the `start_offset` and *end_offset* range.

6.9.2 method_info

The `method_info` structure is defined as follows:

```
method_info {
       method_header_info method_header
       u1 bytecodes[]
}
```

The items in the `method_info` structure are as follows:

`method_header`

The `method_header` item represents either a `method_header_info` or an

`extended_method_header_info` structure:

```
method_header_info {
       u1 bitfield {

         bit[4] flags
         bit[4] max_stack
       }
       u1 bitfield {
         bit[4] nargs
         bit[4] max_locals
       }
}

extended_method_header_info {
       u1 bitfield {
         bit[4] flags
         bit[4] padding
       }
       u1 max_stack
       u1 nargs

       u1 max_locals
}
```

The items of the `method_header_info` and extended_method_header_info structures are as follows:

`flags`

The flags item is a mask of modifiers defined for this method. Valid flag values are shown in the following table.

Flags	Values
ACC_EXTENDED	0x8
ACC_ABSTRACT	0x4

TABLE 6-7 CAP file method flags

The value of the ACC_EXTENDED flag must be one if the `method_header` is represented by an `extended_method_header_info` structure. Otherwise the value must be zero.

The value of the ACC_ABSTRACT flag must be one if this method is defined as abstract. In this case the `bytecodes` array must be empty. If this method is not abstract the value of the ACC_ABSTRACT flag must be zero.

The Java Card virtual machine reserves all other flag values. Their values must be zero.

`padding`

The `padding` item has the value of zero. This item is only defined for the `extended_method_header_info` structure.

`max_stack`

The `max_stack` item indicates the maximum number of 16-bit cells required on the operand stack during execution of this method.

Stack entries of type `int` are represented in two 16-bit cells, while all others are represented in one 16-bit cell.

`nargs`

The `nargs` item indicates the number of 16-bit cells required to represent the parameters passed to this method, including the `this` pointer if this method is a virtual method.

Parameters of type `int` are represented in two 16-bit cells, while all others are represented in one 16-bit cell.

`max_locals`

The max_locals item indicates the number of 16-bit cells required to represent the local variables declared by this method, not including the parameters passed to this method on invocation.[1]

Local variables of type `int` are represented in two 16-bit cells, while all others are represented in one 16-bit cell. The number of cells required for overloaded local variables is two if one or more of the overloaded variables is of type `int`.

`bytecodes[]`

The `bytecodes` item represents an array of Java Card bytecodes that implement this method. Valid instructions are defined in Chapter 7, “Java Card Virtual Machine Instruction Set”. The *impdep1* and *impdep2* bytecodes can not be present in the `bytecodes` array item.

If this method is abstract the `bytecodes` item must contain zero elements.

1. Unlike in Java Card `CAP` files, in Java `class` files the `max_locals` item includes both the local variables declared by the method and the parameters passed to the method.

6.10 Static Field Component

The Static Field Component contains all of the information required to create and initialize an image of all of the static fields defined in this package, referred to as the *static field image*. `Final static` fields of primitive types are not represented in the static field image. Instead these compile-time constants are placed in line in Java Card instructions.

The Static Field Component does not reference any other component in this `CAP` file. The Constant Pool Component (§6.7), Export Component (§6.12) and Descriptor Component (§6.13) reference fields defined in the Static Field Component.

The ordering constraints, or segments, associated with a static field image are shown in TABLE 6-8. `Reference` types occur first in the image. Arrays initialized through Java `<clinit>` methods occur first within the set of `reference` types. Primitive types occur last in the image, and primitive types initialized to non-default values occur last within the set of primitive types.

category	segment	content
`reference` types	1	arrays of primitive types initialized by `<clinit>` methods
	2	`reference` types initialized to `null`
primitive types	3	primitive types initialized to default values
	4	primitive types initialized to non-default values

TABLE 6-8 Segments of a static field image

The number of bytes used to represent each field type in the static field image is shown in the following table.

Type	Bytes
`boolean`	1
`byte`	1
`short`	2
`int`	4
`reference`, including arrays	2

TABLE 6-9 Static field sizes

The `static_field_component` structure is defined as:

```
static_field_component {
       u1 tag
       u2 size
       u2 image_size
       u2 reference_count
       u2 array_init_count
       array_init_info array_init[array_init_count]
       u2 default_value_count
       u2 non_default_value_count
       u1 non_default_values[non_default_values_count]
}
```

The items in the `static_field_component` structure are as follows:

`tag`

The `tag` item has the value `COMPONENT_StaticField` (8).

`size`

The `size` item indicates the number of bytes in the `static_field_component` structure, excluding the `tag` and `size` items. The value of the `size` item must be greater than zero.

`image_size`

The `image_size` item indicates the number of bytes required to represent the static fields defined in this package, excluding final static fields of primitive types. This value is the number of bytes in the static field image. The number of bytes required to represent each field type is shown in TABLE 6-9.

The value of the `image_size` item does not include the number of bytes require to represent the initial values of array instances enumerated in the Static Field Component.

`reference_count`

The `reference_count` item indicates the number of `reference` type static fields defined in this package. This is the number of fields represented in segments 1 and 2 of the static field image as described in TABLE 6-8.

The value of the `reference_count` item may be 0 if no `reference` type fields are defined in this package. Otherwise it must be equal to the number of `reference` type fields defined.

`array_init_count`

The `array_init_count` item indicates the number of elements in the `array_init` array. This is the number of fields represented in segment 1 of the static field image as described in TABLE 6-8. It represents the number of arrays

initialized in all of the <clinit> methods in this package.

If this CAP file defines a library package the value of array_init_count must be zero.

array_init[]

The array_init item represents an array of array_init_info structures that specify the initial array values of static fields of arrays of primitive types. These initial values are indicated in Java <clinit> methods. The array_init_info structure is defined as:

```
array_init_info {
       u1 type
       u2 count
       u1 values[count]
}
```

The items in the array_init_info structure are defined as follows:

type

The type item indicates the type of the primitive array. Valid values are shown in the following table.

Type	Value
boolean	2
byte	3
short	4
int	5

TABLE 6-10 Array types

count

The count item indicates the number of bytes in the values array. It does not represent the number of elements in the static field array (referred to as *length* in Java), since the values array is an array of bytes and the static field array may be a non-byte type. The Java length of the static field array is equal to the count item divided by the number of bytes required to represent the static field type (TABLE 6-9) indicated by the type item.

values

The values item represents a byte array containing the initial values of the static field array. The number of entries in the values array is equal to the size in bytes of the type indicated by the type item. The size in bytes of each type is shown in TABLE 6-9.

`default_value_count`

The `default_value_count` item indicates the number of bytes required to initialize the set of static fields represented in segment 3 of the static field image as described in TABLE 6-8. These static fields are primitive types initialized to default values. The number of bytes required to initialize each static field type is equal to the size in bytes of the type as shown in TABLE 6-9.

`non_default_value_count`

The `non_default_value_count` item represents the number bytes in the `non_default_values` array. This value is equal to the number of bytes in segment 4 of the static field image as described in TABLE 6-8. These static fields are primitive types initialized to non-default values.

`non_default_values[]`

The `non_default_values` item represents an array of bytes of non-default initial values. This is the exact image of segment 4 of the static field image as described in TABLE 6-8. The number of entries in the `non_default_values` array for each static field type is equal to the size in bytes of the type as shown in TABLE 6-9.

6.11 Reference Location Component

The Reference Location Component represents lists of offsets into the `info` item of the Method Component (§6.9) to operands that contain indices into the `constant_pool` array of the Constant Pool Component (§6.7). Some of the constant pool indices are represented in one-byte values while others are represented in two-byte values.

The Reference Location Component is not referenced by any other component in this `CAP` file.

The Reference Location Component structure is defined as:

```
reference_location_component {
       u1 tag
       u2 size
       u2 byte_index_count
       u1 offsets_to_byte_indices[byte_index_count]
       u2 byte2_index_count
       u1 offsets_to_byte2_indices[byte2_index_count]
}
```

The items of the `reference_location_component` structure are as follows:

`tag`

The tag item has the value COMPONENT_ReferenceLocation (9).

`size`

The `size` item indicates the number of bytes in the `reference_location_component` structure, excluding the `tag` and `size` items. The value of the `size` item must be greater than zero.

`byte_index_count`

The `byte_index_count` item represents the number of elements in the `offsets_to_byte_indices` array.

`offsets_to_byte_indices[]`

The `offsets_to_byte_indices` item represents an array of 1-byte jump offsets into the `info` item of the Method Component to each 1-byte `constant_pool` array index. Each entry represents the number of bytes (or *distance*) between the current index to the next. If the distance is greater than or equal to 255 then there are *n* entries equal to 255 in the array, where *n* is equal to the distance divided by 255. The *nth* entry of 255 is followed by an entry containing the value of the distance modulo 255.

An example of the jump offsets in an `offsets_to_byte_indices` array is shown in the following table.

Instruction	Offset to Operand	Jump Offset
`getfield_a 0`	10	10
`putfield_b 2`	65	55
		255
		255
`getfield_s 1`	580	5
		255
`putfield_a 0`	835	0
`getfield_i 3`	843	8

TABLE 6-11 One-byte reference location example

All 1-byte `constant_pool` array indices in the Method Component must be represented in `offsets_to_byte_indices` array.

`byte2_index_count`

The `byte2_index_count` item represents the number of elements in the `offsets_to_byte2_indices` array.

`offsets_to_byte2_indices[]`

The `offsets_to_byte2_indices` item represents an array of 1-byte jump off-

sets into the info item of the Method Component to each 2-byte `constant_pool` array index. Each entry represents the number of bytes (or *distance*) between the current index to the next. If the distance is greater than or equal to 255 then there are *n* entries equal to 255 in the array, where *n* is equal to the distance divided by 255. The *nth* entry of 255 is followed by an entry containing the value of the distance modulo 255.

An example of the jump offsets in an offsets_to_byte_indices array is shown in TABLE 6-11. The same example applies to the offsets_to_byte2_indices array if the instructions are changed to those with 2-byte `constant_pool` array indices.

All 2-byte `constant_pool` array indices in the Method Component must be represented in `offsets_to_byte2_indices` array, including those represented in `catch_type_index` items of the `exception_handler_info` array.

6.12 Export Component

The Export Component lists all static elements in this package that may be imported by classes in other packages. Instance fields and virtual methods are not represented in the Export Component.

If this `CAP` file does not include an Applet Component (§6.5) (called a *library* package) , the Export Component contains an entry for each `public` class and `public` interface defined in this package. Furthermore, for each `public` class there is an entry for each `public` or `protected static` field defined in that class, for each `public` or `protected static` method defined in that class, and for each `public` or protected constructor defined in that class. `Final static` fields of primitive types (compile-time constants) are not included.

If this `CAP` file includes an Applet Component (§6.5) (called an *applet* package) the Export Component includes entries only for all `public` interfaces that are shareable.[1] An interface is sharable if and only if it is the `javacard.framework.Shareable` interface or implements (directly or indirectly) that interface.

Elements in the Export Component reference elements in the Class Component (§6.8), Method Component (§6.9), and Static Field Component (§6.10). No other component in this `CAP` file references the Export Component.

1. The restriction on shareable functionality is imposed by the firewall as defined in the Java Card Runtime Environment (JCRE) 2.1 specification.

The Export Component is represented by the following structure:

```
export_component {
        u1 tag
        u2 size
        u1 class_count
        class_export_info {
                u2 class_offset
                u1 static_field_count
                u1 static_method_count
                u2 static_field_offsets[static_field_count]
                u2 static_method_offsets[static_method_count]
        } class_exports[class_count]
}
```

The items of the `export_component` structure are as follows:

`tag`

The `tag` item has the value `COMPONENT_Export` (10).

`size`

The `size` item indicates the number of bytes in the `export_component` structure, excluding the `tag` and `size` items. The value of the `size` item must be greater than zero.

`class_count`

The `class_count` item represents the number of entries in the `class_exports` table.

`class_exports[]`

The `class_exports` item represents a variable-length table of `class_export_info` structures. If this package is a library package, the table contains an entry for each of the `public` classes and `public` interfaces defined in this package. If this package is an applet package, the table contains an entry for each of the `public` shareable interfaces defined in this package.

An index into the table to a particular class or interface is equal to the token value of that class or interface (§4.3.7.2). The token value is published in the `Export` file (§5.5) of this package.

The items in the `class_export_info` structure are:

`class_offset`

The `class_offset` item represents a byte offset into the `info` item of the Class Component (§6.8). If this package defines a library package, the item at that offset must be either an `interface_info` or a `class_info` structure. The `interface_info` or `class_info` struc-

ture at that offset must represent the exported class or interface.

If this package defines an applet package, the item at the `class_offset` in the `info` item of the Class Component must be an `interface_info` structure. The `interface_info` structure at that offset must represent the exported, shareable interface. In particular, the `ACC_SHAREABLE` flag of the `interface_info` structure must be equal to 1.

`static_field_count`

The `static_field_count` item represents the number of elements in the `static_field_offsets` array. This value indicates the number of `public` and `protected static` fields defined in this class, excluding `final static` fields of primitive types.

If the `class_offset` item represents an offset to an `interface_info` structure, the value of the `static_field_count` item must be zero.

`static_method_count`

The `static_method_count` item represents the number of elements in the `static_method_offsets` array. This value indicates the number of `public` and `protected static` methods and constructors defined in this class.

If the `class_offset` item represents an offset to an `interface_info` structure, the value of the `static_method_count` item must be zero.

`static_field_offsets[]`

The `static_field_offsets` item represents an array of 2-byte offsets into the static field image defined by the Static Field Component (§6.10). Each offset must be to the beginning of the representation of the exported static field.

An index into the `static_field_offsets` array must be equal to the token value of the field represented by that entry. The token value is published in the `Export` file (§5.7) of this package.

`static_method_offsets[]`

The `static_method_offsets` item represents a table of 2-byte offsets into the `info` item of the Method Component (§6.9). Each offset must be to the beginning of a `method_info` structure. The `method_info` structure must represent the exported static method or constructor.

An index into the `static_method_offsets` array must be equal to the token value of the method represented by that entry.

6.13 Descriptor Component

The Descriptor Component provides sufficient information to parse and verify all elements of the CAP file. It references, and therefore describes, elements in the Constant Pool Component (§6.7), Class Component (§6.8), Method Component (§6.9), and Static Field Component (§6.10). No components in the CAP file reference the Descriptor Component.

The Descriptor Component is represented by the following structure:

```
descriptor_component {
       u1 tag
       u2 size
       u1 class_count
       class_descriptor_info classes[class_count]
       type_descriptor_info types
}
```

The items of the descriptor_component structure are as follows:

tag

The tag item has the value COMPONENT_Descriptor (11).

size

The size item indicates the number of bytes in the descriptor_component structure, excluding the tag and size items. The value of the size item must be greater than zero.

class_count

The class_count item represents the number of entries in the classes table.

classes[]

The classes item represents a table of variable-length class_descriptor_info structures. Each class and interface defined in this package is represented in the table.

types

The types item represents a type_descriptor_info structure. This structure lists the set of field types and method signatures of the fields and methods defined or referenced in this package. Those referenced are enumerated in the Constant Pool Component.

6.13.1 class_descriptor_info

The `class_descriptor_info` structure is used to describe a class or interface defined in this package:

```
class_descriptor_info {
       u1 token
       u1 access_flags
       class_ref this_class_ref
       u1 interface_count
       u2 field_count
       u2 method_count
       class_ref interfaces [interface_count]
       field_descriptor_info fields[field_count]
       method_descriptor_info methods[method_count]
}
```

The items of the `class_descriptor_info` structure are as follows:

`token`

The `token` item represents the class token (§4.3.7.2) of this class or interface. If this class or interface is package-visible it does not have a token assigned. In this case the value of the `token` item must be 0xFF.

`access_flags`

The `access_flags` item is a mask of modifiers used to describe the access permission to and properties of this class or interface. The `access_flags` modifiers for classes and interfaces are shown in the following table.

Name	Value
ACC_PUBLIC	0x01
ACC_FINAL	0x10
ACC_INTERFACE	0x40
ACC_ABSTRACT	0x80

TABLE 6-12 CAP file class descriptor flags

The class access and modifier flags defined in the table above are a subset of those defined for classes and interfaces in a Java `class` file. They have the same meaning, and are set under the same conditions, as the corresponding flags in a Java `class` file.

The Java Card virtual machine reserves all other flag values. Their values must be zero.

`this_class_ref`

The `this_class_ref` item is a `class_ref` structure indicating the location of the `class_info` structure in the Class Component (§6.8). The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (§6.7.1).

`interface_count`

The `interface_count` item represents the number of entries in the `interfaces` array.

`field_count`

The `field_count` item represents the number of entries in the `fields` array. If this `class_descriptor_info` structure represents an interface, the value of the `field_count` item is equal to zero.

`method_count`

The `method_count` item represents the number of entries in the `methods` array.

`interfaces[]`

The `interfaces` item represents an array of interfaces implemented by this class or interface. The elements in the array are `class_ref` structures indicating the location of the `class_info` structure in the Class Component (§6.8). The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (§6.7.1).

`fields[]`

The `fields` item represents an array of `field_descriptor_info` structures. Each field declared by this class is represented in the array.

`methods[]`

The `methods` item represents an array of `method_descriptor_info` structures. Each method declared or defined by this class or interface is represented in the array.

6.13.2 field_descriptor_info

The `field_descriptor_info` structure is used to describe a field defined in this package:

```
field_descriptor_info {
       u1 token
       u1 access_flags
       union {
              static_field_ref static_field
              instance_field_ref instance_field
       } field_ref
       union {
              u2 primitive_type
              u2 reference_type
       } type
}
```

The items of the `field_descriptor_info` structure is as follows:

`token`

The `token` item represents the token of this field. If this field is private or package-visible static field it does not have a token assigned. In this case the value of the `token` item must be 0xFF.

`access_flags`

The `access_flags` item is a mask of modifiers used to describe the access permission to and properties of this field. The `access_flags` modifiers for fields are shown in the following table.

Name	Value
ACC_PUBLIC	0x01
ACC_PRIVATE	0x02
ACC_PROTECTED	0x04
ACC_STATIC	0x08
ACC_FINAL	0x10

TABLE 6-13 CAP file field descriptor flagss

The field access and modifier flags defined in the table above are a subset of those defined for fields in a Java `class` file. They have the same meaning, and are set under the same conditions, as the corresponding flags in a Java `class` file.

The Java Card virtual machine reserves all other flag values. Their values must be zero.

`field_ref`

The `field_ref` item represents a reference to this field. If the `ACC_STATIC` flag is equal to 1, this item represents a `static_field_ref` as defined in the `CONSTANT_StaticFieldref` structure (§6.7.3).

If the `ACC_STATIC` flag is equal to 0, this item represents an `instance_field_ref` as defined in the `CONSTANT_InstanceFieldref` structure (§6.7.2).

`type`

The `type` item indicates the type of this field. directly or indirectly. If this field is a primitive type (`boolean`, `byte`, `short`, or `int`) the high bit of this item is equal to 1, otherwise the high bit of this item is equal to 0.

`primitive_type`

The `primitive_type` item represents the type of this field using the values in the table below. As noted above, the high bit of the `primitive_type` item is equal to 1.

Data Type	Value
`boolean`	0x0002
`byte`	0x0003
`short`	0x0004
`int`	0x0005

TABLE 6-14 Primitive type descriptor values

`reference_type`

The `reference_type` item represents a 15-bit offset into the `type_descriptor_info` structure. The item at the offset must represent the `reference` type of this field. As noted above, the high bit of the `reference_type` item is equal to 0.

6.13.3 method_descriptor_info

The `method_descriptor_info` structure is used to describe a method defined in this package:

```
method_descriptor_info {
       u1 token
       u1 access_flags
       u2 method_offset
       u2 type_offset
       u2 bytecode_count
       u2 exception_handler_count
       u2 exception_handler_index
}
```

The items of the `method_descriptor_info` structure are as follows:

`token`

The `token` item represents the static method token (§4.3.7.4) or virtual method token (§4.3.7.6) or interface method token (§4.3.7.7) of this method. If this method is a private or package-visible static method, a private or package-visible constructor, or a private virtual method it does not have a token assigned. In this case the value of the `token` item must be 0xFF.

`access_flags`

The `access_flags` item is a mask of modifiers used to describe the access permission to and properties of this method. The `access_flags` modifiers for methods are shown in the following table.

Name	Value
ACC_PUBLIC	0x01
ACC_PRIVATE	0x02
ACC_PROTECTED	0x04
ACC_STATIC	0x08
ACC_FINAL	0x10
ACC_ABSTRACT	0x40
ACC_INIT	0x80

TABLE 6-15 CAP file method descriptor flags

The method access and modifier flags defined in the table above, except the `ACC_INIT` flag, are a subset of those defined for methods in a Java `class` file. They have the same meaning, and are set under the same conditions, as the corresponding flags in a Java `class` file.

The `ACC_INIT` flag is set if the method descriptor identifies a constructor meth-

ods. In Java a constructor method is recognized by its name, `<init>`, but in Java Card the name is replaced by a token. As in the Java verifier, these methods require special checks by the Java Card verifier.

The Java Card virtual machine reserves all other flag values. Their values must be zero.

`method_offset`

If the `class_descriptor_info` structure that contains this `method_descriptor_info` structure represents a class, the `method_offset` item represents a byte offset into the `info` item of the Method Component (§6.9). The element at that offset must be the beginning of a `method_info` structure. The `method_info` structure must represent this method.

If the `class_descriptor_info` structure that contains this `method_descriptor_info` structure represents an interface, the value of the `method_offset` item must be zero.

`type_offset`

The `type_offset` item must be a valid offset into the `type_descriptor_info` structure. The type described at that offset represents the signature of this method.

`bytecode_count`

The `bytecode_count` item represents the number of bytecodes in this method. The value is equal to the length of the `bytecodes` array item in the `method_info` structure in the method component (§6.9) of this method.

`exception_handler_count`

The `exception_handler_count` item represents the number of exception handlers implemented by this method.

`exception_handler_index`

The `exception_handler_index` item represents the index to the first `exception_handlers` table entry in the method component (§6.9) implemented by this method. Succeeding `exception_handlers` table entries, up to the value of the exception_handler_count item, are also exception handlers implemented by this method.

The value of the `exception_handler_index` item is 0 if the value of the `exception_handler_count` item is 0.

6.13.4 type_descriptor_info

The `type_descriptor_info` structure represents the types of fields and signatures of methods defined in this package:

```
type_descriptor_info {
       u2 constant_pool_count
       u2 constant_pool_types[constant_pool_count]
       { u1 nibble_count;
         u1 type[(nibble_count+1) / 2];
       } type_desc[]
}
```

The `type_descriptor_info` structure contains the following elements:

`constant_pool_count`

The `constant_pool_count` item represents the number of entries in the `constant_pool_types` array. This value is equal to the number of entries in the `constant_pool` array of the Constant Pool Component (§6.7).

`constant_pool_types[]`

The `constant_pool_types` item is an array that describes the types of the fields and methods referenced in the Constant Pool Component. This item has the same number of entries as the `constant_pool` array of the Constant Pool Component, and each entry describes the type of the corresponding entry in the `constant_pool` array.

If the corresponding `constant_pool` array entry represents a class or interface reference, it does not have an associated type. In this case the value of the entry in the `constant_pool_types` array item is 0xFFFF.

If the corresponding `constant_pool` array entry represents a field or method, the value of the entry in the `constant_pool_types` array is an offset into the `type_descriptor_info` structure. The element at that offset must describe the type of the field or the signature of the method.

`type_desc[]`

The `type_desc` item represents a table of variable-length type descriptor structures. These descriptors represent the types of fields and signatures of methods. The elements in the structure are:

`nibble_count`

The `nibble_count` value represents the number of nibbles required to describe the type encoded in the `type` array. This is different from the length of the `type` array if the value of the `nibble_count` item is odd. In this case the length of the `type` array is one greater than the value of `nibble_count`.

`type[]`

The `type` array contains an encoded description of the type, composed of individual nibbles. If the `nibble_count` item is an odd number, the last nibble in the `type` array must be 0x0. The values of the type descriptor nibbles are defined in the following table.

Type	Value
`void`	0x1
`boolean`	0x2
`byte`	0x3
`short`	0x4
`int`	0x5
`reference`	0x6
array of `boolean`	0xA
array of `byte`	0xB
array of `short`	0xC
array of `int`	0xD
array of `reference`	0xE

TABLE 6-16 Type descriptor values

Class `reference` types are described using the `reference` nibble 0x6, followed by a 2-byte (4-nibble) `class_ref` structure. The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (§6.7.1). For example, a field of type `reference` to `p1.c1` in a `CAP` file defining package `p0` is described as:

Nibble	Value	Description
0	0x6	`reference`
1	<p1>	package token (high bit on)
2		
3	<c1>	class token
4		
5	0x0	`padding`

TABLE 6-17 Encoded `reference` type `p1.c1`

The following are examples of the array types:

Nibble	Value	Description
0	0xB	array of `byte`
1	0x0	`padding`

TABLE 6-18 Encoded `byte` array type

Nibble	Value	Description
0	0xE	array of `reference`
1	<p1>	package token (high bit on)
2		
3	<c1>	class token
4		
5	0x0	`padding`

TABLE 6-19 Encoded `reference` array type `p1.c1`

Method signatures are encoded in the same way, with the last nibble indicating the return type of the method. For example:

Nibble	Value	Description
0	0x1	`void`
1	0x0	`padding`

TABLE 6-20 Encoded method signature `()V`

Nibble	Value	Description
0	0x6	`reference`
1	<p1>	package token (high bit on)
2		
3	<c1>	class token
4		
5	0x4	`short`

TABLE 6-21 Encoded method signature `(Lp1.ci;)S`

CHAPTER 7

Java Card Virtual Machine Instruction Set

A Java Card virtual machine instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. This chapter gives details about the format of each Java Card virtual machine instruction and the operation it performs.

7.1 Assumptions: The Meaning of “Must”

The description of each instruction is always given in the context of Java Card virtual machine code that satisfies the static and structural constraints of Chapter 6, “The CAP File Format.”

In the description of individual Java Card virtual machine instructions, we frequently state that some situation “must” or “must not” be the case: “The *value2* must be of type `int`.” The constraints of Chapter 6, “The CAP File Format” guarantee that all such expectations will in fact be met. If some constraint (a “must” or “must not”) in an instruction description is not satisfied at run time, the behavior of the Java Card virtual machine is undefined.

7.2 Reserved Opcodes

In addition to the opcodes of the instructions specified later this chapter, which are used in Java Card `CAP` files (see Chapter 6, “The CAP File Format”), two opcodes are reserved for internal use by a Java Card virtual machine implementation. If Sun extends the instruction set of the Java Card virtual machine in the future, these reserved opcodes are guaranteed not to be used.

The two reserved opcodes, numbers 254 (0xfe) and 255 (0xff), have the mnemonics *impdep1* and *impdep2*, respectively. These instructions are intended to provide “back doors” or traps to implementation-specific functionality implemented in software and hardware, respectively.

Although these opcodes have been reserved, they may only be used inside a Java Card virtual machine implementation. They cannot appear in valid `CAP` files.

7.3 Virtual Machine Errors

A Java Card virtual machine may encounter internal errors or resource limitations that prevent it from executing correctly written Java programs. While the Java Virtual Machine Specification allows reporting and handling of virtual machine errors, it also states that they cannot ordinarily be handled by application code. This Java Card Virtual Machine Specification is more restrictive in that it does not allow for any reporting or handling of unrecoverable virtual machine errors at the application code level. A virtual machine error is considered unrecoverable if further execution could compromise the security or correct operation of the virtual machine or underlying system software. When an unrecoverable error occurs, the virtual machine will halt bytecode execution. Responses beyond halting the virtual machine are implementation-specific policies and are not mandated in this specification.

In the case where the virtual machine encounters a recoverable error, such as insufficient memory to allocate a new object, it will throw a `SystemException` with an error code describing the error condition. The Java Card Virtual Machine Specification cannot predict where resource limitations or internal errors may be encountered and does not mandate precisely when they can be reported. Thus, a `SystemException` may be thrown at any time during the operation of the Java Card virtual machine.

7.4 Security Exceptions

Instructions of the Java Card virtual machine throw an instance of the class `SecurityException` when a security violation has been detected. The Java Card virtual machine does not mandate the complete set of security violations that can or will result in an exception being thrown. However, there is a minimum set that must be supported.

In the general case, any instruction that de-references an object reference must throw a `SecurityException` if the context (§3.4) in which the instruction is executing is different than the owning context (§3.4) of the referenced object. The list of instructions includes the instance field get and put instructions, the array load and store instructions, as well as the *arraylength*, *invokeinterface*, *invokespecial*, *invokevirtual*, *checkcast*, *instanceof* and *athrow* instructions.

There are several exceptions to this general rule that allow cross-context use of objects or arrays. These exceptions are detailed in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*. An important detail to note is that any cross-context method invocation will result in a context switch (§3.4).

The Java Card virtual machine may also throw a `SecurityException` if an instruction violates any of the static constraints of Chapter 6, “The CAP File Format.” The Java Card Virtual Machine Specification does not mandate which instructions must implement these additional security checks, or to what level. Therefore, a `SecurityException` may be thrown at any time during the operation of the Java Card virtual machine.

7.5 The Java Card Virtual Machine Instruction Set

Java Virtual Machine instructions are represented in this chapter by entries of the form shown in the figure below, an example instruction page, in alphabetical order and each beginning on a new page.

mnemonic ***mnemonic***

Short description of the instruction

Format

mnemonic
operand1
operand2
…

Forms

mnemonic = opcode

Stack

…, *value1*, *value2* ⇒
…, *value3*

Description

A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed, the type of the results, etc.

Runtime Exceptions

If any runtime exceptions can be thrown by the execution of an instruction they are set off one to a line, in the order in which they must be thrown.

Other than the runtime exceptions, if any, listed for an instruction, that instruction must not throw any runtime exceptions except for instances of `SystemException`.

Notes

Comments not strictly part of the specification of an instruction are set aside as notes at the end of the description.

FIGURE 7-1 An example instruction page

Each cell in the instruction format diagram represents a single 8-bit byte. The instruction's *mnemonic* is its name. Its opcode is its numeric representation and is given in both decimal and hexadecimal forms. Only the numeric representation is actually present in the Java Card virtual machine code in a `CAP` file.

Keep in mind that there are “operands” generated at compile time and embedded within Java Card virtual machine instructions, as well as “operands” calculated at run time and supplied on the operand stack. Although they are supplied from several different areas, all these operands represent the same thing: values to be operated upon by the Java Card virtual machine instruction being executed. By implicitly taking many of its operands from its operand stack, rather than representing them explicitly in its compiled code as additional operand bytes, register numbers, etc., the Java Card virtual machine’s code stays compact.

Some instructions are presented as members of a family of related instructions sharing a single description, format, and operand stack diagram. As such, a family of instructions includes several opcodes and opcode mnemonics; only the family mnemonic appears in the instruction format diagram, and a separate forms line lists all member mnemonics and opcodes. For example, the forms line for the *sconst_<s>* family of instructions, giving mnemonic and opcode information for the two instructions in that family (*sconst_0* and *sconst_1*), is

Forms *sconst_0* = 3 (0x3),
sconst_1 = 4 (0x4)

In the description of the Java Card virtual machine instructions, the effect of an instruction’s execution on the operand stack (§3.5) of the current frame (§3.5) is represented textually, with the stack growing from left to right and each word represented separately. Thus,

Stack…, *value1*, *value2* ⇒
…, *result*

shows an operation that begins by having a one-word *value2* on top of the operand stack with a one-word *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the operand stack and replaced by a one-word *result*, which has been calculated by the instruction. The remainder of the operand stack, represented by an ellipsis (…), is unaffected by the instruction’s execution.

The type `int` takes two words on the operand stack. In the operand stack representation, each word is represented separately using a dot notation:

Stack…, *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒
…, *result.word1*, *result.word2*

The Java Card Virtual Machine Specification does not mandate how the two words are used to represent the 32-bit `int` value; it only requires that a particular implementation be internally consistent.

Load `reference` from array

Format

aaload

Forms

aaload = 36 (0x24)

Stack

..., arrayref, index ⇒
..., value

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `reference`. The *index* must be of type `short`. Both *arrayref* and *index* are popped from the operand stack. The `reference` *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *aaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aaload* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

In some circumstances, the *aaload* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

Store into `reference` array

Format

aastore

Forms

aastore = 55 (0x37)

Stack

..., arrayref, index, value ⇒
...

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `reference`. The *index* must be of type `short` and the *value* must be of type `reference`. The *arrayref*, *index* and *value* are popped from the operand stack. The `reference` *value* is stored as the component of the array at *index*.

The type of value must be assignment compatible with the type of the components of the array referenced by arrayref. Assignment of a value of reference type *S* (source) to a variable of reference type *T* (target) is allowed only when the type *S* supports all of the operations defined on type *T*. The detailed rules follow:

- If *S* is a class type, then:
  - If *T* is a class type, then *S* must be the same class as *T*, or *S* must be a subclass of *T*;
  - If *T* is an interface type, then *S* must implement interface *T*.
- If *S* is an interface type, then:
  - If *T* is a class type, then *T* must be `Object` (§2.2.2.4);
  - If *T* is an interface type, *T* must be the same interface as *S* or a superinterface of *S*.
- If *S* is an array type[1], namely the type *SC*`[]`, that is, an array of components of type *SC*, then:
  - If *T* is a class type, then *T* must be `Object`.
  - If *T* is an array type, namely the type *TC*`[]`, an array of components of type *TC*, then one of the following must be true:
    - *TC* and *SC* are the same primitive type (§3.1).
    - *TC* and *SC* are reference types (§3.1) with type *SC* assignable to *TC*, by these rules.

1. This version of the Java Card virtual machine specification does not allow for arrays of more than one dimension. Therefore, neither *S* or *T* can be an array type, and the rules for array types do not apply.

- If *T* is an interface type, *T* must be one of the interfaces implemented by arrays[1].

Runtime Exceptions

If *arrayref* is `null`, *aastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *aastore* instruction throws an `ArrayIndexOutOfBoundsException`.

Otherwise, if *arrayref* is not `null` and the actual type of *value* is not assignment compatible with the actual type of the component of the array, *aastore* throws an `ArrayStoreException`.

Notes

In some circumstances, the *aastore* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

1. In the Java Card 2.1 API, arrays do not implement any interfaces. Therefore, *T* cannot be an interface type when *S* is an array type, and this rule does not apply.

Push `null`

Format

aconst_null

Forms

aconst_null = 1 (0x1)

Stack

... ⇒
..., *null*

Description

Push the `null` object `reference` onto the operand stack.

Load `reference` from local variable

Format

aload
index

Forms

aload = 21 (0x15)

Stack

… ⇒
…, *objectref*

Description

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.5). The local variable at *index* must contain a `reference`. The *objectref* in the local variable at *index* is pushed onto the operand stack.

Notes

The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

aload_<n> *aload_<n>*

Load `reference` from local variable

Format

aload_<n>

Forms

aload_0 = 24 (0x18)
aload_1 = 25 (0x19)
aload_2 = 26 (0x1a)
aload_3 = 27 (0x1b)

Stack

… ⇒
…, *objectref*

Description

The <*n*> must be a valid index into the local variables of the current frame (§3.5). The local variable at <*n*> must contain a `reference`. The *objectref* in the local variable at <*n*> is pushed onto the operand stack.

Notes

An *aload_*<*n*> instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore_*<*n*> instruction is intentional.

Each of the *aload_*<*n*> instructions is the same as *aload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

Create new array of `reference`

Format

anewarray
indexbyte1
indexbyte2

Forms

anewarray = 145 (0x91)

Stack

..., count ⇒
..., arrayref

Description

The *count* must be of type `short`. It is popped off the operand stack. The *count* represents the number of components of the array to be created. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The item at that index in the constant pool must be of type `CONSTANT_Classref` (§6.7.1), a reference to a class or interface type. The reference is resolved. A new array with components of that type, of length *count*, is allocated from the heap, and a `reference` *arrayref* to this new array object is pushed onto the operand stack. All components of the new array are initialized to `null`, the default value for `reference` types.

Runtime Exception

If *count* is less than zero, the *anewarray* instruction throws a `NegativeArraySizeException`.

Return `reference` from method

Format

areturn

Forms

areturn = 119 (0x77)

Stack

…, *objectref* ⇒
[empty]

Description

The *objectref* must be of type `reference`. The *objectref* is popped from the operand stack of the current frame (§3.5) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

Get length of array

Format

arraylength

Forms

arraylength = 146 (0x92)

Stack

…, *arrayref* ⇒
…, *length*

Description

The *arrayref* must be of type `reference` and must refer to an array. It is popped from the operand stack. The *length* of the array it references is determined. That *length* is pushed onto the top of the operand stack as a `short`.

Runtime Exception

If *arrayref* is `null`, the *arraylength* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *arraylength* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

Store `reference` into local variable

Format

astore
index

Forms

astore = 40 (0x28)

Stack

…, *objectref* ⇒
…

Description

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.5). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. The *objectref* is popped from the operand stack, and the value of the local variable at *index* is set to *objectref*.

Notes

The *astore* instruction is used with an *objectref* of type `returnAddress` when implementing Java's `finally` keyword. The *aload* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the *astore* instruction is intentional.

Store `reference` into local variable

Format

astore_<n>

Forms

astore_0 = 43 (0x2b)
astore_1 = 44 (0x2c)
astore_2 = 45 (0x2d)
astore_3 = 46 (0x2e)

Stack

…, *objectref* ⇒
…

Description

The <*n*> must be a valid index into the local variables of the current frame (§3.5). The *objectref* on the top of the operand stack must be of type `returnAddress` or of type `reference`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *objectref*.

Notes

An *astore_<n>* instruction is used with an *objectref* of type `returnAddress` when implementing Java's `finally` keyword. An *aload_<n>* instruction cannot be used to load a value of type `returnAddress` from a local variable onto the operand stack. This asymmetry with the corresponding *astore_<n>* instruction is intentional.

Each of the *aload_<n>* instructions is the same as *aload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

athrow *athrow*

Throw exception or error

Format

athrow

Forms

athrow = 147 (0x93)

Stack

…, *objectref* ⇒
objectref

Description

The *objectref* must be of type `reference` and must refer to an object that is an instance of class `Throwable` or of a subclass of `Throwable`. It is popped from the operand stack. The *objectref* is then thrown by searching the current frame (§3.5) for the most recent `catch` clause that catches the class of *objectref* or one of its superclasses.

If a `catch` clause is found, it contains the location of the code intended to handle this exception. The `pc` register is reset to that location, the operand stack of the current frame is cleared, *objectref* is pushed back onto the operand stack, and execution continues. If no appropriate clause is found in the current frame, that frame is popped, the frame of its invoker is reinstated, and the *objectref* is rethrown.

If no `catch` clause is found that handles this exception, the virtual machine exits.

Runtime Exception

If *objectref* is `null`, *athrow* throws a `NullPointerException` instead of *objectref*.

Notes

In some circumstances, the *athrow* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

Load `byte` or `boolean` from array

Format

baload

Forms

baload = 37 (0x25)

Stack

..., arrayref, index ⇒
..., value

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `byte` or of type `boolean`. The *index* must be of type `short`. Both *arrayref* and *index* are popped from the operand stack. The `byte` *value* in the component of the array at *index* is retrieved, sign-extended to a `short` *value*, and pushed onto the top of the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *baload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *baload* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

In some circumstances, the *baload* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

Store into `byte` or `boolean` array

Format

bastore

Forms

bastore = 56 (0x38)

Stack

…, *arrayref*, *index*, *value* ⇒
…

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `byte` or of type `boolean`. The *index* and *value* must both be of type `short`. The *arrayref*, *index* and *value* are popped from the operand stack. The `short` *value* is truncated to a `byte` and stored as the component of the array indexed by *index*.

Runtime Exceptions

If *arrayref* is `null`, *bastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *bastore* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

In some circumstances, the *bastore* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

Push `byte`

Format

bipush
byte

Forms

bipush = 18 (0x12)

Stack

… ⇒
…, *value.word1*, *value.word2*

Description

The immediate *byte* is sign-extended to an `int`, and the resulting *value* is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *bipush* instruction will not be available.

Push `byte`

Format

bspush
byte

Forms

bspush = 16 (0x10)

Stack

... ⇒
..., *value*

Description

The immediate *byte* is sign-extended to a `short`, and the resulting *value* is pushed onto the operand stack.

Check whether object is of given type

Format

checkcast
atype
indexbyte1
indexbyte2

Forms

checkcast = 148 (0x94)

Stack

…, *objectref* ⇒
…, *objectref*

Description

The unsigned byte *atype* is a code that indicates if the type against which the object is being checked is an array type or a class type. It must take one of the following values or zero:

Array Type	atype
T_BOOLEAN	10
T_BYTE	11
T_SHORT	12
T_INT	13
T_REFERENCE	14

If the value of *atype* is 10, 11, 12, or 13, the values of the *indexbyte1* and *indexbyte2* must be zero, and the value of *atype* indicates the array type against which to check the object. Otherwise the unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The item at that index in the constant pool must be of type `CONSTANT_Classref` (§6.7.1), a reference to a class or interface type. The reference is resolved. If the value of *atype* is 14, the object is checked against an array type that is an array of object references of the type of the resolved class. If the value of *atype* is zero, the object is checked against a class or interface type that is the resolved class.

The *objectref* must be of type `reference`. If *objectref* is `null` or can be cast to the specified array type or the resolved class or interface type, the operand stack is unchanged; otherwise the *checkcast* instruction throws a `ClassCastException`.

The following rules are used to determine whether an *objectref* that is not `null` can be cast to the resolved type: if *S* is the class of the object referred to by *objectref* and *T* is

the resolved class, array or interface type, *checkcast* determines whether *objectref* can be cast to type *T* as follows:

- If *S* is a class type, then:
  - If *T* is a class type, then *S* must be the same class as *T*, or *S* must be a subclass of *T*;
  - If *T* is an interface type, then *S* must implement interface *T*.
- If *S* is an interface type, then:
  - If *T* is a class type, then *T* must be `Object` (§2.2.2.4);
  - If *T* is an interface type, *T* must be the same interface as *S* or a superinterface of *S*.
- If *S* is an array type, namely the type *SC*`[]`, that is, an array of components of type *SC*[1], then:
  - If *T* is a class type, then *T* must be `Object`.
  - If *T* is an array type, namely the type *TC*`[]`, an array of components of type *TC*, then one of the following must be true:
    - *TC* and *SC* are the same primitive type (§3.1).
    - *TC* and *SC* are reference types (§3.1) with type *SC* assignable to *TC*, by these rules.
  - If *T* is an interface type, *T* must be one of the interfaces implemented by arrays[2].

Runtime Exception

If *objectref* cannot be cast to the resolved class, array, or interface type, the *checkcast* instruction throws a `ClassCastException`.

Notes

The *checkcast* instruction is fundamentally very similar to the *instanceof* instruction. It differs in its treatment of `null`, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

In some circumstances, the *checkcast* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

---

1. This version of the Java Card virtual machine specification does not allow for arrays of more than one dimension. Therefore, neither *SC* or *TC* can be an array type.
2. In the Java Card 2.1 API, arrays do not implement any interfaces. Therefore, *T* cannot be an interface type when *S* is an array type, and this rule does not apply.

If a virtual machine does not support the `int` data type, the value of *atype* may not be 13 (array type = `T_INT`).

Duplicate top operand stack word

Format

dup

Forms

dup = 61 (0x3d)

Stack

…, *word* ⇒
…, *word, word*

Description

The top word on the operand stack is duplicated and pushed onto the operand stack.

The *dup* instruction must not be used unless *word* contains a 16-bit data type.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the *dup* instruction operates on an untyped word, ignoring the type of data it contains.

Duplicate top operand stack words and insert below

Format

dup_x
mn

Forms

dup_x = 63 (0x3f)

Stack

…, *wordN*, …, *wordM*, …, *word1* ⇒
…, *wordM*, …, *word1*, *wordN*, …, *wordM*, …, *word1*

Description

The unsigned byte *mn* is used to construct two parameter values. The high nibble, (*mn* & 0xf0) >> 4, is used as the value *m*. The low nibble, (*mn* & 0xf), is used as the value *n*. Permissible values for *m* are 1 through 4. Permissible values for *n* are 0 and *m* through *m*+4.

For positive values of *n*, the top *m* words on the operand stack are duplicated and the copied words are inserted *n* words down in the operand stack. When *n* equals 0, the top *m* words are copied and placed on top of the stack.

The *dup_x* instruction must not be used unless the ranges of words 1 through *m* and words *m*+1 through *n* each contain either a 16-bit data type, two 16-bit data types, a 32-bit data type, a 16-bit data type and a 32-bit data type (in either order), or two 32-bit data types.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the *dup_x* instruction operates on untyped words, ignoring the types of data they contain.

If a virtual machine does not support the `int` data type, the permissible values for *m* are 1 or 2, and permissible values for *n* are 0 and *m* through *m*+2.

Duplicate top two operand stack words

Format

dup2

Forms

dup2 = 62 (0x3e)

Stack

…, *word2*, *word1* ⇒
…, *word2*, *word1*, *word2*, *word1*

Description

The top two words on the operand stack are duplicated and pushed onto the operand stack, in the original order.

The *dup2* instruction must not be used unless each of *word1* and *word2* is a word that contains a 16-bit data type or both together are the two words of a single 32-bit datum.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the *dup2* instruction operates on untyped words, ignoring the types of data they contain.

Fetch field from object

Format

getfield_<t>
index

Forms

getfield_a = 131 (0x83)
getfield_b = 132 (0x84)
getfield_s = 133 (0x85)
getfield_i = 134 (0x86)

Stack

…, *objectref* ⇒
…, *value*

OR

…, *objectref* ⇒
…, *value.word1*, *value.word2*

Description

The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *index* is used as an index into the constant pool of the current package (§3.5). The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (§6.7.2), a reference to a class and a field token. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* at that offset into the class instance referenced by *objectref* is fetched. If the *value* is of type `byte` or type `boolean`, it is sign-extended to a short. The *value* is pushed onto the operand stack.

Runtime Exception

If *objectref* is `null`, the *getfield_<t>* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *getfield_<t>* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *getfield_i* instruction will not be available.

Fetch field from current object

Format

getfield_<t>_this
index

Forms

getfield_a_this = 173 (0xad)
getfield_b_this = 174 (0xae)
getfield_s_this = 175 (0xaf)
getfield_i_this = 176 (0xb0)

Stack

… ⇒
…, *value*

OR

… ⇒
…, *value.word1*, *value.word2*

Description

The currently executing method must be an instance method. The local variable at index 0 must contain a `reference` *objectref* to the currently executing method's `this` parameter. The unsigned *index* is used as an index into the constant pool of the current package (§3.5). The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (§6.7.2), a reference to a class and a field token. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* at that offset into the class instance referenced by *objectref* is fetched. If the *value* is of type `byte` or type `boolean`, it is sign-extended to a short. The *value* is pushed onto the operand stack.

Runtime Exception

If *objectref* is `null`, the *getfield_<t>_this* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *getfield_<t>_this* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *getfield_i_this* instruction will not be available.

Fetch field from object (wide index)

Format

getfield_<t>_w
indexbyte1
indexbyte2

Forms

getfield_a_w = 169 (0xa9)
getfield_b_w = 170 (0xaa)
getfield_s_w = 171 (0xab)
getfield_i_w = 172 (0xac)

Stack

…, *objectref* ⇒
…, *value*

OR

…, *objectref* ⇒
…, *value.word1*, *value.word2*

Description

The *objectref*, which must be of type `reference`, is popped from the operand stack. The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (§6.7.2), a reference to a class and a field token. The item must resolve to a field of type `reference`. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* at that offset into the class instance referenced by *objectref* is fetched. If the *value* is of type `byte` or type `boolean`, it is sign-extended to a short. The *value* is pushed onto the operand stack.

Runtime Exception

If *objectref* is `null`, the *getfield_<t>_w* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *getfield_<t>_w* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *getfield_i_w* instruction will not be available.

Get `static` field from class

Format

getstatic_<t>
indexbyte1
indexbyte2

Forms

getstatic_a = 123 (0x7b)
getstatic_b = 124 (0x7c)
getstatic_s = 125 (0x7d)
getstatic_i = 126 (0x7e)

Stack

… ⇒
…, *value*

OR

… ⇒
…, *value.word1*, *value.word2*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item at the index must be of type `CONSTANT_StaticFieldref` (§6.7.3), a reference to a static field. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a class field is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The item is resolved, determining the class field. The *value* of the class field is fetched. If the *value* is of type `byte` or `boolean`, it is sign-extended to a `short`. The *value* is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *getstatic_i* instruction will not be available.

Branch always

Format

goto
branch

Forms

goto = 112 (0x70)

Stack

No change

Description

The value *branch* is used as a signed 8-bit offset. Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

Branch always (wide index)

Format

goto_w
branchbyte1
branchbyte2

Forms

goto_w = 168 (0xa8)

Stack

No change

Description

The unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

Convert `int` to `byte`

Format

i2b

Forms

i2b = 93 (0x5d)

Stack

…, *value.word1*, *value.word2* ⇒
…, *result*

Description

The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack and converted to a `byte` *result* by taking the low-order 16 bits of the `int` value, and discarding the high-order 16 bits. The low-order word is truncated to a `byte`, then sign-extended to a `short` *result*. The *result* is pushed onto the operand stack.

Notes

The *i2b* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

If a virtual machine does not support the `int` data type, the *i2b* instruction will not be available.

Convert `int` to `short`

Format

i2s

Forms

i2s = 94 (0x5e)

Stack

..., value.word1, value.word2 ⇒
..., result

Description

The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack and converted to a `short` *result* by taking the low-order 16 bits of the `int` value and discarding the high-order 16 bits. The *result* is pushed onto the operand stack.

Notes

The *i2s* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

If a virtual machine does not support the `int` data type, the *i2s* instruction will not be available.

Add `int`

Format

iadd

Forms

iadd = 66 (0x42)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

If an i*add* instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide two's-complement format. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Notes

If a virtual machine does not support the `int` data type, the *iadd* instruction will not be available.

Load `int` from array

Format

iaload

Forms

iaload = 39 (0x27)

Stack

..., arrayref, index ⇒
..., value.word1, value.word2

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `int`. The *index* must be of type `short`. Both *arrayref* and *index* are popped from the operand stack. The `int` *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *iaload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *iaload* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

In some circumstances, the *iaload* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *iaload* instruction will not be available.

Boolean AND `int`

Format

iand

Forms

iand = 84 (0x54)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *iand* instruction will not be available.

Store into `int` array

Format

iastore

Forms

iastore = 58 (0x3a)

Stack

…, arrayref, index, value.word1, value.word2 ⇒
…

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `int`. The *index* must be of type `short` and *value* must be of type `int`. The *arrayref*, *index* and *value* are popped from the operand stack. The `int` *value* is stored as the component of the array indexed by *index*.

Runtime Exception

If *arrayref* is `null`, *iastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *iastore* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

In some circumstances, the *iastore* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *iastore* instruction will not be available.

Compare `int`

Format

icmp

Forms

icmp = 95 (0x5f)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result

Description

Both *value1* and *value2* must be of type `int`. They are both popped from the operand stack, and a signed integer comparison is performed. If *value1* is greater than *value2*, the `short` value 1 is pushed onto the operand stack. If *value1* is equal to *value2*, the `short` value 0 is pushed onto the operand stack. If *value1* is less than *value2*, the `short` value –1 is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *icmp* instruction will not be available.

Push `int` constant

Format

iconst_<i>

Forms

iconst_m1 = 10 (0x09)
iconst_0 = 11 (0xa)
i*const_1* = 12 (0xb)
i*const_2* = 13 (0xc)
i*const_3* = 14 (0xd)
i*const_4* = 15 (0xe)
i*const_5* = 16 (0xf)

Stack

… ⇒
…, <*i*>.*word1*, <*i*>.*word2*

Description

Push the `int` constant <*i*> (*-1*, *0*, *1*, *2*, *3*, *4*, or *5*) onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *iconst_<i>* instruction will not be available.

Divide `int`

Format

idiv

Forms

idiv = 72 (0x48)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is the value of the Java expression *value1* / *value2*. The *result* is pushed onto the operand stack.

An `int` division rounds towards 0; that is, the quotient produced for `int` values in *n/d* is an `int` value *q* whose magnitude is as large as possible while satisfying $|d \cdot q| = |n|$. Moreover, *q* is a positive when $|n| = |d|$ and *n* and *d* have the same sign, but *q* is negative when $|n| = |d|$ and *n* and *d* have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of the largest possible magnitude for the `int` type, and the divisor is *–1*, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

Runtime Exception

If the value of the divisor in an `int` division is 0, *idiv* throws an `ArithmeticException`.

Notes

If a virtual machine does not support the `int` data type, the *idiv* instruction will not be available.

Branch if `reference` comparison succeeds

Format

if_acmp<cond>
branch

Forms

if_acmpeq = 104 (0x68)
if_acmpne = 105 (0x69)

Stack

..., value1, value2 ⇒
...

Description

Both *value1* and *value2* must be of type `reference`. They are both popped from the operand stack and compared. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*

If the comparison succeeds, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *if_acmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_acmp<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if_acmp<cond>* instruction.

Branch if `reference` comparison succeeds (wide index)

Format

if_acmp<cond>_w
branchbyte1
branchbyte2

Forms

if_acmpeq_w = 160 (0xa0)
if_acmpne_w = 161 (0xa1)

Stack

…, value1, value2 ⇒
…

Description

Both *value1* and *value2* must be of type `reference`. They are both popped from the operand stack and compared. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1 = value2*
- *ne* succeeds if and only if *value1 ≠ value2*

If the comparison succeeds, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *if_acmp<cond>_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_acmp<cond>_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if_acmp<cond>_w* instruction.

Branch if `short` comparison succeeds

Format

if_scmp<cond>
branch

Forms

if_scmpeq = 106 (0x6a)
if_scmpne = 107 (0x6b)
if_scmplt = 108 (0x6c)
if_scmpge = 109 (0x6d)
if_scmpgt = 110 (0x6e)
if_scmple = 111 (0x6f)

Stack

..., value1, value2 ⇒
...

Description

Both *value1* and *value2* must be of type `short`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*
- *lt* succeeds if and only if *value1* < *value2*
- *le* succeeds if and only if *value1* ≤ *value2*
- *gt* succeeds if and only if *value1* > *value2*
- *ge* succeeds if and only if *value1* ≥ *value2*

If the comparison succeeds, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *if_scmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_scmp<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if_scmp<cond>* instruction.

Branch if `short` comparison succeeds (wide index)

Format

if_scmp<cond>_w
branchbyte1
branchbyte2

Forms

if_scmpeq_w = 162 (0xa2)
if_scmpne_w = 163 (0xa3)
if_scmplt_w = 164 (0xa4)
if_scmpge_w = 165 (0xa5)
if_scmpgt_w = 166 (0xa6)
if_scmple_w = 167 (0xa7)

Stack

…, value1, value2 ⇒
…

Description

Both *value1* and *value2* must be of type `short`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*
- *lt* succeeds if and only if *value1* < *value2*
- *le* succeeds if and only if *value1* ≤ *value2*
- *gt* succeeds if and only if *value1* > *value2*
- *ge* succeeds if and only if *value1* ≥ *value2*

If the comparison succeeds, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *if_scmp<cond>_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_scmp<cond>_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if_scmp<cond>_w* instruction.

Branch if `short` comparison with zero succeeds

Format

if<cond>
branch

Forms

ifeq = 96 (0x60)
ifne = 97 (0x61)
iflt = 98 (0x62)
ifge = 99 (0x63)
ifgt = 100 (0x64)
ifle = 101 (0x65)

Stack

…, *value* ⇒
…

Description

The *value* must be of type `short`. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value* = 0
- *ne* succeeds if and only if *value* ≠ 0
- *lt* succeeds if and only if *value* < 0
- *le* succeeds if and only if *value* ≤ 0
- *gt* succeeds if and only if *value* > 0
- *ge* succeeds if and only if *value* ≥ 0

If the comparison succeeds, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *if<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if<cond>* instruction.

Branch if `short` comparison with zero succeeds (wide index)

Format

if<cond>_w
branchbyte1
branchbyte2

Forms

ifeq_w = 152 (0x98)
ifne_w = 153 (0x99)
iflt_w = 154 (0x9a)
ifge_w = 155 (0x9b)
ifgt_w = 156 (0x9c)
ifle_w = 157 (0x9d)

Stack

…, *value* ⇒
…

Description

The *value* must be of type `short`. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *eq* succeeds if and only if *value* = 0
- *ne* succeeds if and only if *value* ≠ 0
- *lt* succeeds if and only if *value* < 0
- *le* succeeds if and only if *value* ≤ 0
- *gt* succeeds if and only if *value* > 0
- *ge* succeeds if and only if *value* ≥ 0

If the comparison succeeds, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *if<cond>_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if<cond>_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if<cond>_w* instruction.

Branch if `reference` not `null`

Format

ifnonnull
branch

Forms

ifnonnull = 103 (0x67)

Stack

…, *value* ⇒
…

Description

The *value* must be of type `reference`. It is popped from the operand stack. If the *value* is not `null`, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *ifnonnull* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnonnull* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnonnull* instruction.

Branch if `reference` not `null` (wide index)

Format

ifnonnull_w
branchbyte1
branchbyte2

Forms

ifnonnull_w = 159 (0x9f)

Stack

..., value ⇒
...

Description

The *value* must be of type `reference`. It is popped from the operand stack. If the *value* is not `null`, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *ifnonnull_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnonnull_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnonnull_w* instruction.

Branch if `reference` is `null`

Format

ifnull
branch

Forms

ifnull = 102 (0x66)

Stack

…, *value* ⇒
…

Description

The *value* must be of type `reference`. It is popped from the operand stack. If the *value* is `null`, *branch* is used as signed 8-bit offset, and execution proceeds at that offset from the address of the opcode of this *ifnull* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnull* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnull* instruction.

Branch if `reference` is `null` (wide index)

Format

ifnull_w
branchbyte1
branchbyte2

Forms

ifnull_w = 158 (0x9e)

Stack

…, *value* ⇒
…

Description

The *value* must be of type `reference`. It is popped from the operand stack. If the *value* is `null`, the unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *ifnull_w* instruction. The target address must be that of an opcode of an instruction within the method that contains this *ifnull_w* instruction.

Otherwise, execution proceeds at the address of the instruction following this *ifnull_w* instruction.

Increment local `int` variable by constant

Format

iinc
index
const

Forms

iinc = 90 (0x5a)

Stack

No change

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be valid indices into the local variables of the current frame (§3.5). The local variables at *index* and *index* + 1 together must contain an `int`. The *const* is an immediate signed byte. The value *const* is first sign-extended to an `int`, then the `int` contained in the local variables at *index* and *index* + 1 is incremented by that amount.

Notes

If a virtual machine does not support the `int` data type, the *iinc* instruction will not be available.

Increment local `int` variable by constant

Format

iinc_w
index
byte1
byte2

Forms

iinc_w = 151 (0x97)

Stack

No change

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be valid indices into the local variables of the current frame (§3.5). The local variables at *index* and *index* + 1 together must contain an `int`. The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate `short` where the value of the short is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an `int` *const*. The `int` contained in the local variables at *index* and *index* + 1 is incremented by *const*.

Notes

If a virtual machine does not support the `int` data type, the *iinc_w* instruction will not be available.

Push `int`

Format

iipush
byte1
byte2
byte3
byte4

Forms

iipush = 20 (0x14)

Stack

… ⇒
…, *value1.word1*, *value1.word2*

Description

The immediate unsigned *byte1*, *byte2*, *byte3*, and *byte4* values are assembled into a signed `int` where the value of the int is (*byte1* << 24) | (*byte2* << 16) | (*byte3* << 8) | *byte4*. The resulting *value* is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *iipush* instruction will not be available.

Load `int` from local variable

Format

iload
index

Forms

iload = 23 (0x17)

Stack

… ⇒
…, *value1.word1*, *value1.word2*

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be valid indices into the local variables of the current frame (§3.5). The local variables at *index* and *index* + 1 together must contain an `int`. The *value* of the local variables at *index* and *index* + 1 is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *iload* instruction will not be available.

Load `int` from local variable

Format

iload_<n>

Forms

iload_0 = 32 (0x20)
i*load_1* = 33 (0x21)
i*load_2* = 34 (0x22)
i*load_3* = 35 (0x23)

Stack

… ⇒
…, *value1.word1*, *value1.word2*

Description

Both <*n*> and <*n*> + 1 must be a valid indices into the local variables of the current frame (§3.5). The local variables at <*n*> and <*n*> + 1 together must contain an `int`. The *value* of the local variables at <*n*> and <*n*> + 1 is pushed onto the operand stack.

Notes

Each of the *iload_<n>* instructions is the same as *iload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

If a virtual machine does not support the `int` data type, the *iload_<n>* instruction will not be available.

Access jump table by key match and jump

Format

ilookupswitch
defaultbyte1
defaultbyte2
npairs1
npairs2
match-offset pairs…

Pair Format

matchbyte1
matchbyte2
matchbyte3
matchbyte4
offsetbyte1
offsetbyte2

Forms

ilookupswitch = 118 (0x76)

Stack

…, *key.word1*, *key.word2* ⇒
…

Description

An *ilookupswitch* instruction is a variable-length instruction. Immediately after the *ilookupswitch* opcode follow a signed 16-bit value *default*, an unsigned 16-bit value *npairs*, and then *npairs* pairs. Each pair consists of an `int` *match* and a signed 16-bit *offset*. Each *match* is constructed from four unsigned bytes as (*matchbyte1* << 24) | (*matchbyte2* << 16) | (*matchbyte3* << 8) | *matchbyte4*. Each *offset* is constructed from two unsigned bytes as (*offsetbyte1* << 8) | *offsetbyte2*.

The table *match-offset* pairs of the *ilookupswitch* instruction must be sorted in increasing numerical order by *match*.

The *key* must be of type `int` and is popped from the operand stack and compared against the *match* values. If it is equal to one of them, then a target address is calculated by adding the corresponding *offset* to the address of the opcode of this *ilookupswitch* instruction. If the *key* does not match any of the *match* values, the target address is calculated by adding *default* to the address of the opcode of this *ilookupswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from the offset of each *match-offset* pair, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *ilookupswitch* instruction.

Notes

The *match-offset* pairs are sorted to support lookup routines that are quicker than linear search.

If a virtual machine does not support the `int` data type, the *ilookupswitch* instruction will not be available.

Multiply `int`

Format

imul

Forms

imul = 70 (0x46)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

If an *imul* instruction overflows, then the result is the low-order bits of the mathematical product as an `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two values.

Notes

If a virtual machine does not support the `int` data type, the *imul* instruction will not be available.

Negate `int`

Format

ineg

Forms

ineg = 76 (0x4c)

Stack

..., value.word1, value.word2 ⇒
..., result.word1, result.word2

Description

The *value* must be of type `int`. It is popped from the operand stack. The `int` *result* is the arithmetic negation of *value*, *-value*. The *result* is pushed onto the operand stack.

For `int` values, negation is the same as subtraction from zero. Because the Java Card virtual machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `int` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all `int` values `x`, `-x` equals `(~x) + 1`.

Notes

If a virtual machine does not support the `int` data type, the *imul* instruction will not be available.

Determine if object is of given type

Format

instanceof
atype
indexbyte1
indexbyte2

Forms

instanceof = 149 (0x95)

Stack

…, *objectref* ⇒
…, *result*

Description

The unsigned byte *atype* is a code that indicates if the type against which the object is being checked is an array type or a class type. It must take one of the following values or zero:

Array Type	atype
`T_BOOLEAN`	`10`
`T_BYTE`	`11`
`T_SHORT`	`12`
`T_INT`	`13`
`T_REFERENCE`	`14`

If the value of *atype* is 10, 11, 12, or 13, the values of the *indexbyte1* and *indexbyte2* must be zero, and the value of *atype* indicates the array type against which to check the object. Otherwise the unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The item at that index in the constant pool must be of type `CONSTANT_Classref` (§6.7.1), a reference to a class or interface type. The reference is resolved. If the value of *atype* is 14, the object is checked against an array type that is an array of object references of the type of the resolved class. If the value of *atype* is zero, the object is checked against a class or interface type that is the resolved class.

The *objectref* must be of type `reference`. It is popped from the operand stack. If *objectref* is not `null` and is an instance of the resolved class, array or interface, the *instanceof* instruction pushes a `short` *result* of *1* on the operand stack. Otherwise it pushes a `short` *result* of *0*.

The following rules are used to determine whether an *objectref* that is not `null` is an instance of the resolved type: if *S* is the class of the object referred to by *objectref* and *T* is the resolved class, array or interface type, *instanceof* determines whether *objectref* is an instance of *T* as follows:

- If *S* is a class type, then:
  - If *T* is a class type, then *S* must be the same class as *T*, or *S* must be a subclass of *T*;
  - If *T* is an interface type, then *S* must implement interface *T*.
- If *S* is an interface type, then:
  - If *T* is a class type, then *T* must be `Object` (§2.2.2.4);
  - If *T* is an interface type, *T* must be the same interface as *S* or a superinterface of *S*.
- If *S* is an array type, namely the type *SC*`[]`, that is, an array of components of type *SC*[1], then:
  - If *T* is a class type, then *T* must be `Object`.
  - If *T* is an array type, namely the type *TC*`[]`, an array of components of type *TC*, then one of the following must be true:
    - *TC* and *SC* are the same primitive type (§3.1).
    - *TC* and *SC* are reference types (§3.1) with type *SC* assignable to *TC*, by these rules.
  - If *T* is an interface type, *T* must be one of the interfaces implemented by arrays[2].

Notes

The *instanceof* instruction is fundamentally very similar to the *checkcast* instruction. It differs in its treatment of `null`, its behavior when its test fails (*checkcast* throws an exception, *instanceof* pushes a result code), and its effect on the operand stack.

In some circumstances, the *instanceof* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the value of *atype* may not be 13 (array type = `T_INT`).

---

1. This version of the Java Card virtual machine specification does not allow for arrays of more than one dimension. Therefore, neither *SC* or *TC* can be an array type.
2. In the Java Card 2.1 API, arrays do not implement any interfaces. Therefore, *T* cannot be an interface type when *S* is an array type, and this rule does not apply.

Invoke interface method

Format

invokeinterface
nargs
indexbyte1
indexbyte2
method

Forms

invokeinterface = 142 (0x8e)

Stack

…, *objectref*, [*arg1*, [*arg2* …]] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item at that index must be of type `CONSTANT_Classref` (§6.7.1), a reference to an interface class. The specified method is resolved. The interface method must not be `<init>`, an instance initialization method, or `<clinit>`, a class or interface initialization method.

The *nargs* operand is an unsigned byte that must not be zero. The *method* operand is an unsigned byte that is the interface method token for the method to be invoked. The *objectref* must be of type `reference` and must be followed on the operand stack by *nargs* – 1 words of arguments. The number of words of arguments and the type and order of the values they represent must be consistent with those of the selected interface method.

The interface table of the class of the type of *objectref* is determined. If *objectref* is an array type, then the interface table of class `Object` (§2.2.2.4) is used. The interface table is searched for the resolved interface. The result of the search is a table that is used to map the *method* token to a *index*.

The *index* is an unsigned byte that is used as an index into the method table of the class of the type of *objectref*. If the *objectref* is an array type, then the method table of class `Object` is used. The table entry at that index includes a direct reference to the method's code and modifier information.

The *nargs* – 1 words of arguments and *objectref* are popped from the operand stack. A new stack frame is created for the method being invoked, and *objectref* and the arguments are made the values of its first *nargs* words of local variables, with *arg1* in local variable at index 0, *arg1* in local variable at offset 2, *arg2* immediately following

that, and so on. The new stack frame is then made current, and the Java Card virtual machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

Runtime Exception

If *objectref* is `null`, the *invokeinterface* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *invokeinterface* instruction may throw a `SecurityException` if the current context (§3.4) is not the context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*. If the current context is not the object's context and the JCRE permits invocation of the method, the *invokeinterface* instruction will cause a context switch (§3.4) to the object's context before invoking the method, and will cause a return context switch to the previous context when the invoked method returns.

invokespecial *invokespecial*

Invoke instance method; special handling for superclass, private, and instance initialization method invocations

Format

invokespecial
indexbyte1
indexbyte2

Forms

invokespecial = 140 (0x8c)

Stack

…, *objectref*, [*arg1*, [*arg2* …]] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an *index* into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. If the invoked method is a private instance method or an instance initialization method, the constant pool item at *index* must be of type `CONSTANT_StaticMethodref` (§6.7.3), a reference to a statically linked instance method. If the invoked method is a superclass method, the constant pool item at *index* must be of type `CONSTANT_SuperMethodref` (§6.7.2), a reference to an instance method of a specified class. The reference is resolved. The resolved method must not be `<clinit>`, a class or interface initialization method. If the method is `<init>`, an instance initialization method, then the method must only be invoked once on an uninitialized object, and before the first backward branch following the execution of the *new* instruction that allocated the object. Finally, if the method is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The resolved method includes the code for the method, an unsigned byte *nargs* that must not be zero, and the method's modifier information.

The *objectref* must be of type `reference`, and must be followed on the operand stack by *nargs* – 1 words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the selected instance method.

The *nargs* – 1 words of arguments and *objectref* are popped from the operand stack. A new stack frame is created for the method being invoked, and *objectref* and the arguments are made the values of its first *nargs* words of local variables, with *objectref* in local variable *0*, *arg1* in local variable *1*, and so on. The new stack frame is

then made current, and the Java Card virtual machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

Runtime Exception

If *objectref* is `null`, the *invokespecial* instruction throws a `NullPointerException`.

Invoke a class (`static`) method

Format

invokestatic
indexbyte1
indexbyte2

Forms

invokestatic = 141 (0x8d)

Stack

…, [*arg1*, [*arg2* …]] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item at that index must be of type `CONSTANT_StaticMethodref` (§6.7.3), a reference to a static method. The method must not be `<init>`, an instance initialization method, or `<clinit>`, a class or interface initialization method. It must be `static`, and therefore cannot be `abstract`. Finally, if the method is `protected`, then it must be either a member of the current class or a member of a superclass of the current class.

The resolved method includes the code for the method, an unsigned byte *nargs* that may be zero, and the method's modifier information.

The operand stack must contain *nargs* words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the resolved method .

The *nargs* words of arguments are popped from the operand stack. A new stack frame is created for the method being invoked, and the words of arguments are made the values of its first *nargs* words of local variables, with *arg1* in local variable *0*, *arg2* in local variable *1*, and so on. The new stack frame is then made current, and the Java Card virtual machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

Invoke instance method; dispatch based on class

Format

invokevirtual
indexbyte1
indexbyte2

Forms

invokevirtual = 139 (0x8b)

Stack

…, *objectref*, [*arg1*, [*arg2* …]] ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item at that index must be of type `CONSTANT_VirtualMethodref` (§6.7.2), a reference to a class and a virtual method token. The specified method is resolved. The method must not be `<init>`, an instance initialization method, or `<clinit>`, a class or interface initialization method. If the method is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The resolved method reference includes an unsigned *index* into the method table of the resolved class and an unsigned byte *nargs* that must not be zero.

The *objectref* must be of type `reference`. The *index* is an unsigned byte that is used as an index into the method table of the class of the type of *objectref*. If the *objectref* is an array type, then the method table of class `Object` (§2.2.2.4) is used. The table entry at that index includes a direct reference to the method's code and modifier information.

The *objectref* must be followed on the operand stack by *nargs* – 1 words of arguments, where the number of words of arguments and the type and order of the values they represent must be consistent with those of the selected instance method.

The *nargs* – 1 words of arguments and *objectref* are popped from the operand stack. A new stack frame is created for the method being invoked, and *objectref* and the arguments are made the values of its first *nargs* words of local variables, with *objectref* in local variable *0*, *arg1* in local variable *1*, and so on. The new stack frame is then made current, and the Java Card virtual machine `pc` is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

Runtime Exception

If *objectref* is `null`, the *invokevirtual* instruction throws a `NullPointerException`.

In some circumstances, the *invokevirtual* instruction may throw a `SecurityException` if the current context (§3.4) is not the context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*. If the current context is not the object's context and the JCRE permits invocation of the method, the *invokevirtual* instruction will cause a context switch (§3.4) to the object's context before invoking the method, and will cause a return context switch to the previous context when the invoked method returns.

Boolean OR `int`

Format

ior

Forms

ior = 86 (0x56)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *ior* instruction will not be available.

Remainder `int`

Format

irem

Forms

irem = 74 (0x4a)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is the value of the Java expression *value1* – (*value1* / *value2*) * *value2*. The *result* is pushed onto the operand stack.

The result of the *irem* instruction is such that `(a/b)*b + (a%b)` is equal to `a`. This identity holds even in the special case that the dividend is the negative `int` of largest possible magnitude for its type and the divisor is –`1` (the remainder is `0`). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

Runtime Exception

If the value of the divisor for a `short` remainder operator is 0, *irem* throws an `ArithmeticException`.

Notes

If a virtual machine does not support the `int` data type, the *irem* instruction will not be available.

Return `int` from method

Format

ireturn

Forms

ireturn = 121 (0x79)

Stack

…, *value.word1*, *value.word2* ⇒
[empty]

Description

The *value* must be of type `int`. It is popped from the operand stack of the current frame (§3.5) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

Notes

If a virtual machine does not support the `int` data type, the *ireturn* instruction will not be available.

Shift left `int`

Format

ishl

Forms

ishl = 78 (0x4e)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* left by *s* bit positions, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

Notes

This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

If a virtual machine does not support the `int` data type, the *ishl* instruction will not be available.

Arithmetic shift right `int`

Format

ishr

Forms

ishr = 80 (0x50)

Stack

…, *value1.word1*, *value1.word2*, *value2.word1*, *value2.word2* ⇒
…, *result.word1*, *result.word2*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* right by *s* bit positions, with sign extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

Notes

The resulting value is $\lfloor (value1) / 2^{s} \rfloor$, where *s* is *value2* & 0x1f. For nonnegative *value1*, this is equivalent (even if overflow occurs) to truncating `int` division by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

Notes

If a virtual machine does not support the `int` data type, the *ishr* instruction will not be available.

Store `int` into local variable

Format

istore
index

Forms

istore = 42 (0x2a)

Stack

..., value.word1, value.word2 ⇒
...

Description

The *index* is an unsigned byte. Both *index* and *index* + 1 must be a valid index into the local variables of the current frame (§3.5). The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack, and the local variables at *index* and *index* + 1 are set to *value*.

Notes

If a virtual machine does not support the `int` data type, the *istore* instruction will not be available.

Store `int` into local variable

Format

istore_<n>

Forms

istore_0 = 51 (0x33)
i*store_1* = 52 (0x34)
i*store_2* = 53 (0x35)
i*store_3* = 54 (0x36)

Stack

…, value.word1, value.word2 ⇒
…

Description

Both <*n*> and <*n*> + 1 must be a valid indices into the local variables of the current frame (§3.5). The *value* on top of the operand stack must be of type `int`. It is popped from the operand stack, and the local variables at *index* and *index* + 1 are set to *value.*

Notes

If a virtual machine does not support the `int` data type, the *istore_<n>* instruction will not be available.

Subtract `int`

Format

isub

Forms

isub = 68 (0x44)

Stack

..., value1.word1, value1.word2, value2.word1, value2.word2 ⇒
..., result.word1, result.word2

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

For `int` subtraction, `a – b` produces the same result as `a + (–b)`. For `int` values, subtraction from zeros is the same as negation.

Despite the fact that overflow or underflow may occur, in which case the *result* may have a different sign than the true mathematical result, execution of an *isub* instruction never throws a runtime exception.

Notes

If a virtual machine does not support the `int` data type, the *isub* instruction will not be available.

Access jump table by `int` index and jump

Format

itableswitch
defaultbyte1
defaultbyte2
lowbyte1
lowbyte2
lowbyte3
lowbyte4
highbyte1
highbyte2
highbyte3
highbyte4
jump offsets…

Offset Format

offsetbyte1
offsetbyte2

Forms

itableswitch = 116 (0x74)

Stack

…, index ⇒
…

Description

An *itableswitch* instruction is a variable-length instruction. Immediately after the *itableswitch* opcode follow a signed 16-bit value *default*, a signed 32-bit value *low*, a signed 32-bit value *high*, and then *high* – *low* + 1 further signed 16-bit offsets. The value *low* must be less than or equal to *high*. The *high* – *low* + 1 signed 16-bit offsets are treated as a 0-based jump table. Each of the signed 16-bit values is constructed from two unsigned bytes as (*byte1* << 8) | *byte2*. Each of the signed 32-bit values is constructed from four unsigned bytes as (*byte1* << 24) | (*byte2* << 16) | (*byte3* << 8) | *byte4*.

The *index* must be of type `int` and is popped from the stack. If *index* is less than *low* or *index* is greater than *high*, then a target address is calculated by adding *default* to the address of the opcode of this *itableswitch* instruction. Otherwise, the offset at position *index* – *low* of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *itableswitch* instruction. Execution then continues at the target address.

The target addresses that can be calculated from each jump table offset, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *itableswitch* instruction.

Notes

If a virtual machine does not support the `int` data type, the *itableswitch* instruction will not be available.

Logical shift right `int`

Format

iushr

Forms

iushr = 82 (0x52)

Stack

…, *value1.word1, value1.word2, value2.word1, value2.word2* ⇒
…, *result.word1, result.word2*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting the result right by *s* bit positions, with zero extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

Notes

If *value1* is positive and *s* is *value2* & 0x1f, the result is the same as that of *value1* >> *s*; if *value1* is negative, the result is equal to the value of the expression (*value1* >> *s*) + (2 << ~*s*). The addition of the (2 << ~*s*) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

If a virtual machine does not support the `int` data type, the *iushr* instruction will not be available.

Boolean XOR `int`

Format

ixor

Forms

ixor = 88 (0x58)

Stack

…, *value1.word1, value1.word2, value2.word1, value2.word2* ⇒
…, *result.word1, result.word2*

Description

Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *ixor* instruction will not be available.

Jump subroutine

Format

jsr
branchbyte1
branchbyte2

Forms

jsr = 113 (0x71)

Stack

… ⇒
…, *address*

Description

The *address* of the opcode of the instruction immediately following this *jsr* instruction is pushed onto the operand stack as a value of type `returnAddress`. The unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of this *jsr* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr* instruction.

Notes

The *jsr* instruction is used with the *ret* instruction in the implementation of the `finally` clause of the Java language. Note that *jsr* pushes the address onto the stack and *ret* gets it out of a local variable. This asymmetry is intentional.

Create new object

Format

new
indexbyte1
indexbyte2

Forms

new = 143 (0x8f)

Stack

… ⇒
…, *objectref*

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The item at that index in the constant pool must be of type `CONSTANT_Classref` (§6.7.1), a reference to a class or interface type. The reference is resolved and must result in a class type (it must not result in an interface type). Memory for a new instance of that class is allocated from the heap, and the instance variables of the new object are initialized to their default initial values. The *objectref*, a `reference` to the instance, is pushed onto the operand stack.

Notes

The *new* instruction does not completely create a new instance; instance creation is not completed until an instance initialization method has been invoked on the uninitialized instance.

Create new array

Format

newarray
atype

Forms

newarray = 144 (0x90)

Stack

…, *count* ⇒
…, *arrayref*

Description

The *count* must be of type `short`. It is popped off the operand stack. The *count* represents the number of elements in the array to be created.

The unsigned byte *atype* is a code that indicates the type of array to create. It must take one of the following values:

Array Type	*atype*
`T_BOOLEAN`	`10`
`T_BYTE`	`11`
`T_SHORT`	`12`
`T_INT`	`13`

A new array whose components are of type *atype*, of length *count*, is allocated from the heap. A `reference` *arrayref* to this new array object is pushed onto the operand stack. All of the elements of the new array are initialized to the default initial value for its type.

Runtime Exception

If *count* is less than zero, the *newarray* instruction throws a `NegativeArraySizeException`.

Notes

If a virtual machine does not support the `int` data type, the value of *atype* may not be 13 (array type = `T_INT`).

nop *nop*

Do nothing

Format

nop

Forms

nop = 0 (0x0)

Stack

No change

Description

Do nothing.

Pop top operand stack word

Format

pop

Forms

pop = 59 (0x3b)

Stack

…, *word* ⇒
…

Description

The top word is popped from the operand stack.

Notes

The *pop* instruction operates on an untyped word, ignoring the type of data it contains.

pop2 *pop2*

Pop top two operand stack words

Format

pop2

Forms

pop2 = 60 (0x3c)

Stack

…, *word2*, *word1* ⇒
…

Description

The top two words are popped from the operand stack.

The *pop2* instruction must not be used unless each of *word1* and *word2* is a word that contains a 16-bit data type or both together are the two words of a single 32-bit datum.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the *pop2* instruction operates on an untyped word, ignoring the type of data it contains.

Set field in object

Format

putfield_<t>
index

Forms

putfield_a = 135 (0x87)
putfield_b = 136 (0x88)
putfield_s = 137 (0x89)
putfield_i = 138 (0x8a)

Stack

…, *objectref*, *value* ⇒
…

OR

…, *objectref*, *value.word1*, *value.word2* ⇒
…

Description

The unsigned *index* is used as an index into the constant pool of the current package (§3.5). The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (§6.7.2), a reference to a class and a field token. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *objectref*, which must be of type `reference`, and the *value* are popped from the operand stack. If the field is of type `byte` or type `boolean`, the *value* is truncated to a `byte`. The field at the offset from the start of the object referenced by *objectref* is set to the *value*.

Runtime Exception

If *objectref* is `null`, the *putfield_<t>* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *putfield_<t>* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *putfield_i* instruction will not be available.

Set field in current object

Format

putfield_<t>_this
index

Forms

putfield_a_this = 181 (0xb5)
putfield_b_this = 182 (0xb6)
putfield_s_this = 183 (0xb7)
putfield_i_this = 184 (0xb8)

Stack

..., value ⇒
...

OR

..., value.word1, value.word2 ⇒
...

Description

The currently executing method must be an instance method that was invoked using the *invokevirtual*, *invokeinterface* or *invokespecial* instruction. The local variable at index 0 must contain a `reference` *objectref* to the currently executing method's `this` parameter. The unsigned *index* is used as an index into the constant pool of the current package (§3.5). The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (§6.7.2), a reference to a class and a field token. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *value* is popped from the operand stack. If the field is of type `byte` or type `boolean`, the *value* is truncated to a `byte`. The field at the offset from the start of the object referenced by *objectref* is set to the *value*.

Runtime Exception

If *objectref* is `null`, the *putfield_<t>_this* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *putfield_<t>_this* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *putfield_i_this* instruction will not be available.

Set field in object (wide index)

Format

putfield<t>_w
indexbyte1
indexbyte2

Forms

putfield_a_w = 177 (0xb1)
putfield_b_w = 178 (0xb2)
putfield_s_w = 179 (0xb3)
putfield_i_w = 180 (0xb4)

Stack

…, *objectref*, *value* ⇒
…

OR

…, *objectref*, *value.word1*, *value.word2* ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (§6.7.2), a reference to a class and a field token. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class, and the class of *objectref* must be either the current class or a subclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The *objectref*, which must be of type `reference`, and the *value* are popped from the operand stack. If the field is of type `byte` or type `boolean`, the *value* is truncated to a `byte`. The field at the offset from the start of the object referenced by *objectref* is set to the *value*.

Runtime Exception

If *objectref* is `null`, the *putfield_<t>_w* instruction throws a `NullPointerException`.

Notes

In some circumstances, the *putfield_<t>_w* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object referenced by *objectref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

If a virtual machine does not support the `int` data type, the *putfield_i_w* instruction will not be available.

Set `static` field in class

Format

putstatic_<t>
indexbyte1
indexbyte2

Forms

putstatic_a = 127 (0x7f)
putstatic_b = 128 (0x80)
putstatic_s = 129 (0x81)
putstatic_i = 130 (0x82)

Stack

…, value ⇒
…

OR

…, value.word1, value.word2 ⇒
…

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current package (§3.5), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The constant pool item at the index must be of type `CONSTANT_StaticFieldref` (§6.7.3), a reference to a static field. If the field is `protected`, then it must be either a member of the current class or a member of a superclass of the current class.

The item must resolve to a field with a type that matches *t*, as follows:

- *a* field must be of type `reference`
- *b* field must be of type `byte` or type `boolean`
- *s* field must be of type `short`
- *i* field must be of type `int`

The width of a class field is determined by the field type specified in the instruction. The item is resolved, determining the class field. The *value* is popped from the operand stack. If the field is of type `byte` or type `boolean`, the *value* is truncated to a `byte`. The field is set to the *value*.

Notes

In some circumstances, the *putstatic_a* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the object being stored in the field. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification.*

If a virtual machine does not support the `int` data type, the *putstatic_i* instruction will not be available.

Return from subroutine

Format

ret
index

Forms

ret = 114 (0x72)

Stack

No change

Description

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.5). The local variable at *index* must contain a value of type `returnAddress`. The contents of the local variable are written into the Java Card virtual machine's `pc` register, and execution continues there.

Notes

The *ret* instruction is used with the *jsr* instruction in the implementation of the `finally` keyword of the Java language. Note that *jsr* pushes the address onto the stack and *ret* gets it out of a local variable. This asymmetry is intentional.

The *ret* instruction should not be confused with the *return* instruction. A *return* instruction returns control from a Java method to its invoker, without passing any value back to the invoker.

return *return*

Return `void` from method

Format

return

Forms

return = 122 (0x7a)

Stack

... ⇒
[empty]

Description

Any values on the operand stack of the current method are discarded. The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

Convert `short` to `byte`

Format

s2b

Forms

s2b = 91 (0x5b)

Stack

…, *value* ⇒
…, *result*

Description

The *value* on top of the operand stack must be of type `short`. It is popped from the top of the operand stack, truncated to a `byte` *result*, then sign-extended to a `short` *result*. The *result* is pushed onto the operand stack.

Notes

The *s2b* instruction performs a narrowing primitive conversion. It may lose information about the overall magnitude of *value*. The *result* may also not have the same sign as *value*.

Convert `short` to `int`

Format

s2i

Forms

s2i = 92 (0x5c)

Stack

..., value ⇒
..., result.word1, result.word2

Description

The *value* on top of the operand stack must be of type `short`. It is popped from the operand stack and sign-extended to an `int` *result*. The *result* is pushed onto the operand stack.

Notes

The *s2i* instruction performs a widening primitive conversion. Because all values of type `short` are exactly representable by type `int`, the conversion is exact.

If a virtual machine does not support the `int` data type, the *s2i* instruction will not be available.

Add `short`

Format

sadd

Forms

sadd = 65 (0x41)

Stack

..., value1, value2 ⇒
..., result

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is *value1 + value2*. The *result* is pushed onto the operand stack.

If a *sadd* instruction overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide two's-complement format. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Load `short` from array

Format

saload

Forms

saload = 38 (0x46)

Stack

..., arrayref, index ⇒
..., value

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `short`. The *index* must be of type `short`. Both *arrayref* and *index* are popped from the operand stack. The `short` *value* in the component of the array at *index* is retrieved and pushed onto the top of the operand stack.

Runtime Exceptions

If *arrayref* is `null`, *saload* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *saload* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

In some circumstances, the *saload* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

Boolean AND `short`

Format

sand

Forms

sand = 83 (0x53)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

Both *value1* and *value2* are popped from the operand stack. A `short` *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.

Store into `short` array

Format

sastore

Forms

sastore = 57 (0x39)

Stack

…, *arrayref*, *index*, *value* ⇒
…

Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `short`. The *index* and *value* must both be of type `short`. The *arrayref*, *index* and *value* are popped from the operand stack. The `short` *value* is stored as the component of the array indexed by *index*.

Runtime Exception

If *arrayref* is `null`, *sastore* throws a `NullPointerException`.

Otherwise, if *index* is not within the bounds of the array referenced by *arrayref*, the *sastore* instruction throws an `ArrayIndexOutOfBoundsException`.

Notes

In some circumstances, the *sastore* instruction may throw a `SecurityException` if the current context (§3.4) is not the owning context (§3.4) of the array referenced by *arrayref*. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Java Card 2.1 Runtime Environment (JCRE) Specification*.

Push `short` constant

Format

sconst_<s>

Forms

sconst_m1 = 2 (0x2)
sconst_0 = 3 (0x3)
sconst_1 = 4 (0x4)
sconst_2 = 5 (0x5)
sconst_3 = 6 (0x6)
sconst_4= 7 (0x7)
sconst_5 = 8 (0x8)

Stack

… ⇒
…, <*s*>

Description

Push the `short` constant <*s*> (*-1*, *0*, *1*, *2*, *3*, *4*, or *5*) onto the operand stack.

Divide `short`

Format

sdiv

Forms

sdiv = 71 (0x47)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is the value of the Java expression *value1* / *value2*. The *result* is pushed onto the operand stack.

A `short` division rounds towards 0; that is, the quotient produced for `short` values in *n/d* is a `short` value *q* whose magnitude is as large as possible while satisfying | *d* · *q* | = | *n* |. Moreover, *q* is a positive when | *n* | = | *d* | and *n* and *d* have the same sign, but *q* is negative when | *n* | = | *d* | and *n* and *d* have opposite signs.

There is one special case that does not satisfy this rule: if the dividend is the negative integer of the largest possible magnitude for the `short` type, and the divisor is *–1*, then overflow occurs, and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case.

Runtime Exception

If the value of the divisor in a `short` division is 0, *sdiv* throws an `ArithmeticException`.

Increment local `short` variable by constant

Format

sinc
index
const

Forms

sinc = 89 (0x59)

Stack

No change

Description

The *index* is an unsigned byte that must be a valid index into the local variable of the current frame (§3.5). The *const* is an immediate signed byte. The local variable at *index* must contain a `short`. The value *const* is first sign-extended to a `short`, then the local variable at *index* is incremented by that amount.

Increment local `short` variable by constant

Format

sinc_w
index
byte1
byte2

Forms

sinc_w = 150 (0x96)

Stack

No change

Description

The *index* is an unsigned byte that must be a valid index into the local variable of the current frame (§3.5). The immediate unsigned *byte1* and *byte2* values are assembled into a `short` *const* where the value of *const* is (*byte1* << 8) | *byte2*. The local variable at *index*, which must contain a `short`, is incremented by *const*.

Push `short`

Format

sipush
byte1
byte2

Forms

sipush = 19 (0x13)

Stack

… ⇒
…, *value1.word1*, *value1.word2*

Description

The immediate unsigned *byte1* and *byte2* values are assembled into a signed `short` where the value of the short is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an `int`, and the resulting *value* is pushed onto the operand stack.

Notes

If a virtual machine does not support the `int` data type, the *sipush* instruction will not be available.

Load `short` from local variable

Format

sload
index

Forms

sload = 22 (0x16)

Stack

… ⇒
…, *value*

Description

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.5). The local variable at *index* must contain a `short`. The *value* in the local variable at *index* is pushed onto the operand stack.

Load `short` from local variable

Format

sload_<n>

Forms

sload_0 = 28 (0x1c)
sload_1 = 29 (0x1d)
sload_2 = 30 (0x1e)
sload_3 = 31 (0x1f)

Stack

... ⇒
..., *value*

Description

The <*n*> must be a valid index into the local variables of the current frame (§3.5). The local variable at <*n*> must contain a `short`. The *value* in the local variable at <*n*> is pushed onto the operand stack.

Notes

Each of the *sload_<n>* instructions is the same as *sload* with an *index* of <*n*>, except that the operand <*n*> is implicit.

Access jump table by key match and jump

Format

slookupswitch
defaultbyte1
defaultbyte2
npairs1
npairs2
match-offset pairs…

Pair Format

matchbyte1
matchbyte2
offsetbyte1
offsetbyte2

Forms

slookupswitch = 117 (0x75)

Stack

…, *key* ⇒
…

Description

A *slookupswitch* instruction is a variable-length instruction. Immediately after the *slookupswitch* opcode follow a signed 16-bit value *default*, an unsigned 16-bit value *npairs*, and then *npairs* pairs. Each pair consists of a `short` *match* and a signed 16-bit *offset*. Each of the signed 16-bit values is constructed from two unsigned bytes as (*byte1* << 8) | *byte2*.

The table *match-offset* pairs of the *slookupswitch* instruction must be sorted in increasing numerical order by *match*.

The *key* must be of type `short` and is popped from the operand stack and compared against the *match* values. If it is equal to one of them, then a target address is calculated by adding the corresponding *offset* to the address of the opcode of this *slookupswitch* instruction. If the *key* does not match any of the *match* values, the target address is calculated by adding *default* to the address of the opcode of this *slookupswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from the offset of each *match-offset* pair, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *slookupswitch* instruction.

Notes

The *match-offset* pairs are sorted to support lookup routines that are quicker than linear search.

Multiply `short`

Format

smul

Forms

smul = 69 (0x45)

Stack

..., value1, value2 ⇒
..., result

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is *value1* * *value2*. The *result* is pushed onto the operand stack.

If a *smul* instruction overflows, then the result is the low-order bits of the mathematical product as a `short`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical product of the two values.

Negate `short`

Format

sneg

Forms

sneg = 72 (0x4b)

Stack

…, *value* ⇒
…, *result*

Description

The *value* must be of type `short`. It is popped from the operand stack. The `short` *result* is the arithmetic negation of *value*, *-value*. The *result* is pushed onto the operand stack.

For `short` values, negation is the same as subtraction from zero. Because the Java Card virtual machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `short` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all `short` values `x`, `-x` equals `(~x) + 1`.

Boolean OR `short`

Format

sor

Forms

sor = 85 (0x55)

Stack

..., value1, value2 ⇒
..., result

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A `short` *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

Remainder `short`

Format

srem

Forms

srem = 73 (0x49)

Stack

..., value1, value2 ⇒
..., result

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is the value of the Java expression *value1* – (*value1* / *value2*) * *value2*. The *result* is pushed onto the operand stack.

The result of the *irem* instruction is such that `(a/b)*b + (a%b)` is equal to `a`. This identity holds even in the special case that the dividend is the negative `short` of largest possible magnitude for its type and the divisor is –`1` (the remainder is `0`). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative and can be positive only if the dividend is positive. Moreover, the magnitude of the result is always less than the magnitude of the divisor.

Runtime Exception

If the value of the divisor for a `short` remainder operator is 0, *srem* throws an `ArithmeticException`.

Return `short` from method

Format

sreturn

Forms

sreturn = 120 (0x78)

Stack

…, *value* ⇒
[empty]

Description

The *value* must be of type `short`. It is popped from the operand stack of the current frame (§3.5) and pushed onto the operand stack of the frame of the invoker. Any other values on the operand stack of the current method are discarded.

The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

Shift left `short`

Format

sshl

Forms

sshl = 77 (0x4d)

Stack

..., value1, value2 ⇒
..., result

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A `short` *result* is calculated by shifting *value1* left by *s* bit positions, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

Notes

This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

Arithmetic shift right `short`

Format

sshr

Forms

sshr = 79 (0x4f)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A `short` *result* is calculated by shifting *value1* right by *s* bit positions, with sign extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

Notes

The resulting value is $\lfloor (value1) / 2^{s} \rfloor$, where *s* is *value2* & 0x1f. For nonnegative *value1*, this is equivalent (even if overflow occurs) to truncating `short` division by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

Push `short`

Format

sspush
byte1
byte2

Forms

sspush = 17 (0x11)

Stack

… ⇒
…, *value*

Description

The immediate unsigned *byte1* and *byte2* values are assembled into a signed `short` where the value of the short is (*byte1* << 8) | *byte2*. The resulting *value* is pushed onto the operand stack.

Store `short` into local variable

Format

sstore
index

Forms

sstore = 41 (0x29)

Stack

..., value ⇒
...

Description

The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.5). The *value* on top of the operand stack must be of type `short`. It is popped from the operand stack, and the value of the local variable at *index* is set to *value*.

Store `short` into local variable

Format

sstore_<n>

Forms

sstore_0 = 47 (0x2f)
sstore_1 = 48 (0x30)
sstore_2 = 49 (0x31)
sstore_3 = 50 (0x32)

Stack

..., value ⇒
...

Description

The <*n*> must be a valid index into the local variables of the current frame (§3.5). The *value* on top of the operand stack must be of type `short`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *value.*

Subtract `short`

Format

ssub

Forms

ssub = 67 (0x43)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. The `short` *result* is *value1* - *value2*. The *result* is pushed onto the operand stack.

For `short` subtraction, `a – b` produces the same result as `a + (–b)`. For `short` values, subtraction from zeros is the same as negation.

Despite the fact that overflow or underflow may occur, in which case the *result* may have a different sign than the true mathematical result, execution of a *ssub* instruction never throws a runtime exception.

Access jump table by `short` index and jump

Format

stableswitch
defaultbyte1
defaultbyte2
lowbyte1
lowbyte2
highbyte1
highbyte2
jump offsets…

Offset Format

offsetbyte1
offsetbyte2

Forms

stableswitch = 115 (0x73)

Stack

…, *index* ⇒
…

Description

A *stableswitch* instruction is a variable-length instruction. Immediately after the *stableswitch* opcode follow a signed 16-bit value *default*, a signed 16-bit value *low*, a signed 16-bit value *high*, and then *high* – *low* + 1 further signed 16-bit offsets. The value *low* must be less than or equal to *high*. The *high* – *low* + 1 signed 16-bit offsets are treated as a 0-based jump table. Each of the signed 16-bit values is constructed from two unsigned bytes as (*byte1* << 8) | *byte2*.

The *index* must be of type `short` and is popped from the stack. If *index* is less than *low* or *index* is greater than *high*, than a target address is calculated by adding *default* to the address of the opcode of this *stableswitch* instruction. Otherwise, the offset at position *index* – *low* of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *stableswitch* instruction. Execution then continues at the target address.

The target addresses that can be calculated from each jump table offset, as well as the one calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *stableswitch* instruction.

Logical shift right `short`

Format

sushr

Forms

sushr = 81 (0x51)

Stack

..., value1, value2 ⇒
..., result

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A `short` *result* is calculated by sign-extending *value1* to 32 bits and shifting the result right by *s* bit positions, with zero extension, where *s* is the value of the low five bits of *value2*. The resulting value is then truncated to a 16-bit *result*. The *result* is pushed onto the operand stack.

Notes

If *value1* is positive and *s* is *value2* & 0x1f, the result is the same as that of *value1* >> *s*; if *value1* is negative, the result is equal to the value of the expression (*value1* >> *s*) + (2 << ~*s*). The addition of the (2 << ~*s*) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

Swap top two operand stack words

Format

swap_x
mn

Forms

swap_x = 64 (0x40)

Stack

…, *wordM+N*, …, *wordM+1*, *wordM*, …, *word1* ⇒
…, *wordM*, …, *word1*, *wordM+N*, …, *wordM+1*

Description

The unsigned byte *mn* is used to construct two parameter values. The high nibble, (*mn* & 0xf0) >> 4, is used as the value *m*. The low nibble, (*mn* & 0xf), is used as the value *n*. Permissible values for both *m* and *n* are 1 and 2.

The top *m* words on the operand stack are swapped with the *n* words immediately below.

The *swap_x* instruction must not be used unless the ranges of words 1 through *m* and words *m*+1 through *n* each contain either a 16-bit data type, two 16-bit data types, a 32-bit data type, a 16-bit data type and a 32-bit data type (in either order), or two 32-bit data types.

Notes

Except for restrictions preserving the integrity of 32-bit data types, the *swap_x* instruction operates on untyped words, ignoring the types of data they contain.

If a virtual machine does not support the `int` data type, the only permissible value for both *m* and *n* is 1.

Boolean XOR `short`

Format

sxor

Forms

sxor = 87 (0x57)

Stack

…, *value1*, *value2* ⇒
…, *result*

Description

Both *value1* and *value2* must be of type `short`. The values are popped from the operand stack. A `short` *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

CHAPTER 8

Tables of Instructions

TABLE 8-1 Instructions by Opcode Value

dec	hex	mnemonic	dec	hex	mnemonic
0	00	nop	47	2F	sstore_0
1	01	aconst_null	48	30	sstore_1
2	02	sconst_m1	49	31	sstore_2
3	03	sconst_0	50	32	sstore_3
4	04	sconst_1	51	33	istore_0
5	05	sconst_2	52	34	istore_1
6	06	sconst_3	53	35	istore_2
7	07	sconst_4	54	36	istore_3
8	08	sconst_5	55	37	aastore
9	09	iconst_m1	56	38	bastore
10	0A	iconst_0	57	39	sastore
11	0B	iconst_1	58	3A	iastore
12	0C	iconst_2	59	3B	pop
13	0D	iconst_3	60	3C	pop2
14	0E	iconst_4	61	3D	dup
15	0F	iconst_5	62	3E	dup2
16	10	bspush	63	3F	dup_x
17	11	sspush	64	40	swap_x
18	12	bipush	65	41	sadd
19	13	sipush	66	42	iadd
20	14	iipush	67	43	ssub
21	15	aload	68	44	isub
22	16	sload	69	45	smul
23	17	iload	70	46	imul
24	18	aload_0	71	47	sdiv
25	19	aload_1	72	48	idiv
26	1A	aload_2	73	49	srem
27	1B	aload_3	74	4A	irem
28	1C	sload_0	75	4B	sneg
29	1D	sload_1	76	4C	ineg
30	1E	sload_2	77	4D	sshl
31	1F	sload_3	78	4E	ishl
32	20	iload_0	79	4F	sshr
33	21	iload_1	80	50	ishr
34	22	iload_2	81	51	sushr
35	23	iload_3	82	52	iushr
36	24	aaload	83	53	sand
37	25	baload	84	54	iand
38	26	saload	85	55	sor
39	27	iaload	86	56	ior
40	28	astore	87	57	sxor
41	29	sstore	88	58	ixor
42	2A	istore	89	59	sinc
43	2B	astore_0	90	5A	iinc
44	2C	astore_1	91	5B	s2b
45	2D	astore_2	92	5C	s2i
46	2E	astore_3	93	5D	i2b

Table 8-1 (continued) Instructions by Opcode Value

dec	hex	mnemonic	dec	hex	mnemonic
94	5E	i2s	141	8D	invokestatic
95	5F	icmp	142	8E	invokeinterface
96	60	ifeq	143	8F	new
97	61	ifne	144	90	newarray
98	62	iflt	145	91	anewarray
99	63	ifge	146	92	arraylength
100	64	ifgt	147	93	athrow
101	65	ifle	148	94	checkcast
102	66	ifnull	149	95	instanceof
103	67	ifnonnull	150	96	sinc_w
104	68	if_acmpeq	151	97	iinc_w
105	69	if_acmpne	152	98	ifeq_w
106	6A	if_scmpeq	153	99	ifne_w
107	6B	if_scmpne	154	9A	iflt_w
108	6C	if_scmplt	155	9B	ifge_w
109	6D	if_scmpge	156	9C	ifgt_w
110	6E	if_scmpgt	157	9D	ifle_w
111	6F	if_scmple	158	9E	ifnull_w
112	70	goto	159	9F	ifnonnull_w
113	71	jsr	160	A0	if_acmpeq_w
114	72	ret	161	A1	if_acmpne_w
115	73	stableswitch	162	A2	if_scmpeq_w
116	74	itableswitch	163	A3	if_scmpne_w
117	75	slookupswitch	164	A4	if_scmplt_w
118	76	ilookupswitch	165	A5	if_scmpge_w
119	77	areturn	166	A6	if_scmpgt_w
120	78	sreturn	167	A7	if_scmple_w
121	79	ireturn	168	A8	goto_w
122	7A	return	169	A9	getfield_a_w
123	7B	getstatic_a	170	AA	getfield_b_w
124	7C	getstatic_b	171	AB	getfield_s_w
125	7D	getstatic_s	172	AC	getfield_i_w
126	7E	getstatic_i	173	AD	getfield_a_this
127	7F	putstatic_a	174	AE	getfield_b_this
128	80	putstatic_b	175	AF	getfield_s_this
129	81	putstatic_s	176	B0	getfield_i_this
130	82	putstatic_i	177	B1	putfield_a_w
131	83	getfield_a	178	B2	putfield_b_w
132	84	getfield_b	179	B3	putfield_s_w
133	85	getfield_s	180	B4	putfield_i_w
134	86	getfield_i	181	B5	putfield_a_this
135	87	putfield_a	182	B6	putfield_b_this
136	88	putfield_b	183	B7	putfield_s_this
137	89	putfield_s	184	B8	putfield_i_this
138	8A	putfield_i			…
139	8B	invokevirtual	254	FE	impdep1
140	8C	invokespecial	255	FF	impdep2

TABLE 8-2 Instructions by Opcode Mnemonic

mnemonic	dec	hex	mnemonic	dec	hex
aaload	36	24	iand	84	54
aastore	55	37	iastore	58	3A
aconst_null	1	01	icmp	95	5F
aload	21	15	iconst_0	10	0A
aload_0	24	18	iconst_1	11	0B
aload_1	25	19	iconst_2	12	0C
aload_2	26	1A	iconst_3	13	0D
aload_3	27	1B	iconst_4	14	0E
anewarray	145	91	iconst_5	15	0F
areturn	119	77	iconst_m1	9	09
arraylength	146	92	idiv	72	48
astore	40	28	if_acmpeq	104	68
astore_0	43	2B	if_acmpeq_w	160	A0
astore_1	44	2C	if_acmpne	105	69
astore_2	45	2D	if_acmpne_w	161	A1
astore_3	46	2E	if_scmpeq	106	6A
athrow	147	93	if_scmpeq_w	162	A2
baload	37	25	if_scmpge	109	6D
bastore	56	38	if_scmpge_w	165	A5
bipush	18	12	if_scmpgt	110	6E
bspush	16	10	if_scmpgt_w	166	A6
checkcast	148	94	if_scmple	111	6F
dup	61	3D	if_scmple_w	167	A7
dup_x	63	3F	if_scmplt	108	6C
dup2	62	3E	if_scmplt_w	164	A4
getfield_a	131	83	if_scmpne	107	6B
getfield_a_this	173	AD	if_scmpne_w	163	A3
getfield_a_w	169	A9	ifeq	96	60
getfield_b	132	84	ifeq_w	152	98
getfield_b_this	174	AE	ifge	99	63
getfield_b_w	170	AA	ifge_w	155	9B
getfield_i	134	86	ifgt	100	64
getfield_i_this	176	B0	ifgt_w	156	9C
getfield_i_w	172	AC	ifle	101	65
getfield_s	133	85	ifle_w	157	9D
getfield_s_this	175	AF	iflt	98	62
getfield_s_w	171	AB	iflt_w	154	9A
getstatic_a	123	7B	ifne	97	61
getstatic_b	124	7C	ifne_w	153	99
getstatic_i	126	7E	ifnonnull	103	67
getstatic_s	125	7D	ifnonnull_w	159	9F
goto	112	70	ifnull	102	66
goto_w	168	A8	ifnull_w	158	9E
i2b	93	5D	iinc	90	5A
i2s	94	5E	iinc_w	151	97
iadd	66	42	iipush	20	14
iaload	39	27	iload	23	17

Table 8-2 (continued) Instructions by Opcode Mnemonic

mnemonic	dec	hex	mnemonic	dec	hex
iload_0	32	20	putstatic_s	129	81
iload_1	33	21	ret	114	72
iload_2	34	22	return	122	7A
iload_3	35	23	s2b	91	5B
ilookupswitch	118	76	s2i	92	5C
imul	70	46	sadd	65	41
ineg	76	4C	saload	38	26
instanceof	149	95	sand	83	53
invokeinterface	142	8E	sastore	57	39
invokespecial	140	8C	sconst_0	3	03
invokestatic	141	8D	sconst_1	4	04
invokevirtual	139	8B	sconst_2	5	05
ior	86	56	sconst_3	6	06
irem	74	4A	sconst_4	7	07
ireturn	121	79	sconst_5	8	08
ishl	78	4E	sconst_m1	2	02
ishr	80	50	sdiv	71	47
istore	42	2A	sinc	89	59
istore_0	51	33	sinc_w	150	96
istore_1	52	34	sipush	19	13
istore_2	53	35	sload	22	16
istore_3	54	36	sload_0	28	1C
isub	68	44	sload_1	29	1D
itableswitch	116	74	sload_2	30	1E
iushr	82	52	sload_3	31	1F
ixor	88	58	slookupswitch	117	75
jsr	113	71	smul	69	45
new	143	8F	sneg	75	4B
newarray	144	90	sor	85	55
nop	0	00	srem	73	49
pop	59	3B	sreturn	120	78
pop2	60	3C	sshl	77	4D
putfield_a	135	87	sshr	79	4F
putfield_a_this	181	B5	sspush	17	11
putfield_a_w	177	B1	sstore	41	29
putfield_b	136	88	sstore_0	47	2F
putfield_b_this	182	B6	sstore_1	48	30
putfield_b_w	178	B2	sstore_2	49	31
putfield_i	138	8A	sstore_3	50	32
putfield_i_this	184	B8	ssub	67	43
putfield_i_w	180	B4	stableswitch	115	73
putfield_s	137	89	sushr	81	51
putfield_s_this	183	B7	swap_x	64	40
putfield_s_w	179	B3	sxor	87	57
putstatic_a	127	7F			
putstatic_b	128	80			
putstatic_i	130	82			

Glossary

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

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

Applet developer refers to a person creating a Java Card applet using the Java Card technology specifications.

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.

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.

Context is the object space partition associated with a package. Applets within the same Java package belong to the same context. The firewall is the boundary between contexts (see Current context).

Current context. The JCRE keeps track of the current Java Card context. When a virtual method is invoked on an object, and a context switch is required and permitted, the current context is changed to correspond to the context of the applet that owns the object. When that method returns, the previous context is restored. Invocations of static methods have no effect on the current context. The current context and sharing status of an object together determine if access to an object is permissible.

Firewall is the mechanism in the Java Card technology by which the Java Card VM prevents an applet in one context from making unauthorized accesses to objects owned by an applet in another context or the JCRE context, and reports or otherwise addresses the violation.

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 installing applets.

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

Instance variables (also known as *fields*) represent a portion of an object's internal state. Each object has its own set of instance variables. Objects of the same class will have the same instance variables, but each object can have different values.

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.

JAR is an acronym for Java Archive. JAR is a platform-independent file format that combines many files into one.

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

JCRE implementer refers to a person creating a vendor-specific framework using the Java Card 2.1 API.

JCVM is an acronym for the Java Card Virtual Machine. The JCVM executes byte code and manages classes and objects. It enforces separation between applets (firewalls) and enables secure data sharing.

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 programming methodology 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 namespace within the Java programming language and can have classes and interfaces. A package is the smallest unit within the Java programming language.