Java Card™ Platform

Virtual Machine Specification, Classic Edition

Version 3.1

February 2021
Third-Party Content, Products, and Services Disclaimer

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.
Contents

Preface ............................................................................................................................................. 19
Who Should Use This Specification ............................................................................................. 19
Before You Read This Specification ............................................................................................ 19
Shell Prompts ................................................................................................................................... 19
Typographic Conventions ............................................................................................................... 20
Related Documentation .................................................................................................................. 20
Third-Party Web Sites .................................................................................................................... 20
Documentation Accessibility ........................................................................................................... 21
Access to Oracle Support ............................................................................................................... 21
Oracle Welcomes Your Comments ................................................................................................ 21

1 Introduction .................................................................................................................................... 22
1.1 Motivation .................................................................................................................................. 22
1.2 The Java Card Virtual Machine ................................................................................................ 23
1.3 Java Language Security ............................................................................................................ 25
1.4 Java Card Runtime Environment Security ................................................................................. 25

2 A Subset of the Java Virtual Machine .............................................................................................. 27
2.1 Why a Subset is Needed ............................................................................................................ 27
2.2 Java Card Platform Language Subset ....................................................................................... 27
2.2.1 Unsupported Items ................................................................................................................ 27
2.2.1.1 Unsupported Features ....................................................................................................... 27
2.2.1.1.1 Dynamic Class Loading ................................................................................................ 27
2.2.1.1.2 Security Manager ......................................................................................................... 28
2.2.1.1.3 Finalization .................................................................................................................. 28
2.2.1.1.4 Threads ......................................................................................................................... 28
2.2.1.1.5 Cloning ......................................................................................................................... 28
2.2.1.1.6 Access Control in Java Packages ............................................................................... 28
2.2.1.1.7 Typesafe Enums ......................................................................................................... 28
2.2.1.1.8 Enhanced for Loop ..................................................................................................... 29
2.2.1.1.9 Varargs .......................................................... 29
2.2.1.1.10 Runtime Visible Metadata (Annotations) .................. 29
2.2.1.1.11 Assertions ....................................................... 29
2.2.1.2 Unsupported Keywords ........................................... 29
2.2.1.3 Unsupported Types ................................................ 29
2.2.1.4 Unsupported Classes .............................................. 29
   2.2.1.4.1 System ......................................................... 30
2.2.2 Supported Items ...................................................... 30
   2.2.2.1 Supported Features ............................................ 30
      2.2.2.1.1 Packages .................................................. 30
      2.2.2.1.2 Dynamic Object Creation ............................... 30
      2.2.2.1.3 Virtual Methods ....................................... 30
      2.2.2.1.4 Interfaces ............................................... 30
      2.2.2.1.5 Exceptions ............................................... 30
      2.2.2.1.6 Generics ................................................ 31
      2.2.2.1.7 Static Import .......................................... 31
      2.2.2.1.8 Runtime Invisible Metadata (Annotations) .......... 31
      2.2.2.2 Supported Keywords ...................................... 31
      2.2.2.3 Supported Types .......................................... 32
      2.2.2.4 Supported Classes ....................................... 32
         2.2.2.4.1 Object ............................................... 32
         2.2.2.4.2 Throwable .......................................... 32
2.2.3 Optionally Supported Items ........................................ 32
   2.2.3.1 Integer Data Type .......................................... 33
   2.2.3.2 Object Deletion Mechanism ................................ 33
2.2.4 Limitations of the Java Card Virtual Machine .................. 33
   2.2.4.1 Limitations of Packages ................................... 33
      2.2.4.1.1 Packages in a Java Card CAP file .................... 33
      2.2.4.1.2 Package References .................................. 33
      2.2.4.1.3 Package Name ........................................ 33
   2.2.4.2 Limitations of Classes ...................................... 34
      2.2.4.2.1 Classes in a Package ................................ 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.4.2.2 Interfaces ......................................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.2.3 Static Fields ..................................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.2.4 Static Methods ................................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.3 Limitations of Objects ..................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.3.1 Methods ........................................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.3.2 Class Instances ................................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.3.3 Arrays ..........................................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.4 Limitations of Methods ...................................................</td>
<td>34</td>
</tr>
<tr>
<td>2.2.4.5 Limitations of Switch Statements ......................................</td>
<td>35</td>
</tr>
<tr>
<td>2.2.4.6 Limitations of Class Initialization ....................................</td>
<td>35</td>
</tr>
<tr>
<td>2.2.5 Multiselectable Applets Restrictions ....................................</td>
<td>35</td>
</tr>
<tr>
<td>2.2.6 Java Card Platform Remote Method Invocation (RMI) Restrictions ...</td>
<td>35</td>
</tr>
<tr>
<td>2.2.6.1 Remote Classes and Remote Interfaces ..................................</td>
<td>36</td>
</tr>
<tr>
<td>2.2.6.2 Access Control of Remote Interfaces ....................................</td>
<td>36</td>
</tr>
<tr>
<td>2.2.6.3 Parameters and Return Values ..........................................</td>
<td>36</td>
</tr>
<tr>
<td>2.3 Java Card VM Subset ...................................................................</td>
<td>36</td>
</tr>
<tr>
<td>2.3.1 Class File Subset ...................................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.1 Not Supported in Class Files ...........................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.1.1 Class Access Flags .....................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.1.2 Field Descriptors ......................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.1.3 Constant Pool ................................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.1.4 Fields .........................................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.1.5 Methods ........................................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.2 Supported in Class Files ..................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.2.1 ClassFile ......................................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.2.2 Field Descriptors ......................................................</td>
<td>37</td>
</tr>
<tr>
<td>2.3.1.2.3 Method Descriptors .....................................................</td>
<td>38</td>
</tr>
<tr>
<td>2.3.1.2.4 Constant Pool ................................................................</td>
<td>38</td>
</tr>
<tr>
<td>2.3.1.2.5 Fields .........................................................................</td>
<td>38</td>
</tr>
<tr>
<td>2.3.1.2.6 Methods .......................................................................</td>
<td>38</td>
</tr>
<tr>
<td>2.3.1.2.7 Attributes ....................................................................</td>
<td>38</td>
</tr>
<tr>
<td>2.3.2 Bytecode Subset .....................................................................</td>
<td>38</td>
</tr>
</tbody>
</table>
2.3.2.1 Unsupported Bytecodes ............................................................... 38
2.3.2.2 Supported Bytecodes ................................................................. 40
2.3.2.3 Static Restrictions on Bytecodes ................................................. 42
   2.3.2.3.1 ldc, ldc_w ................................................................. 42
   2.3.2.3.2 lookupswitch ............................................................... 42
   2.3.2.3.3 tableswitch ................................................................ 42
   2.3.2.3.4 wide ............................................................................. 42
2.3.3 Exceptions ................................................................................. 43
   2.3.3.1 Uncaught and Uncatchable Exceptions .................................... 43
   2.3.3.2 Checked Exceptions ............................................................. 43
   2.3.3.3 Runtime Exceptions ............................................................ 44
   2.3.3.4 Errors .............................................................................. 44
3 Structure of the Java Card Virtual Machine ............................................ 46
   3.1 Data Types and Values ................................................................. 46
   3.2 Words ...................................................................................... 46
   3.3 Runtime Data Areas .................................................................. 47
   3.4 Contexts .................................................................................. 47
   3.5 Frames .................................................................................... 47
   3.6 Representation of Objects ......................................................... 48
   3.7 Special Initialization Methods ................................................... 48
   3.8 Exceptions .............................................................................. 48
   3.9 Binary File Formats ................................................................. 48
   3.10 Instruction Set Summary ......................................................... 48
      3.10.1 Types and the Java Card Virtual Machine .............................. 49
4 Binary Representation ....................................................................... 51
   4.1 Java Card Platform File Formats ............................................ 51
      4.1.1 Export File Format .......................................................... 51
      4.1.2 CAP File Format ............................................................ 52
      4.1.3 JAR File Container .......................................................... 52
   4.2 AID-based Naming ................................................................ 53
      4.2.1 The AID Format ............................................................... 53
      4.2.2 AID Usage ..................................................................... 53
7.5.7 areturn ............................................................................................................. 165
7.5.8 arraylength ..................................................................................................... 166
7.5.9 astore ............................................................................................................... 166
7.5.10 astore_<n> .................................................................................................. 167
7.5.11 athrow .......................................................................................................... 168
7.5.12 baload .......................................................................................................... 169
7.5.13 bastore ......................................................................................................... 169
7.5.14 bipush .......................................................................................................... 170
7.5.15 bspush ......................................................................................................... 171
7.5.16 checkcast ..................................................................................................... 171
7.5.17 dup ............................................................................................................... 173
7.5.18 dup_x .......................................................................................................... 174
7.5.19 dup2 ............................................................................................................ 175
7.5.20 getfield_<t> ............................................................................................... 175
7.5.21 getfield_<t>_this ......................................................................................... 176
7.5.22 getfield_<t>_w ............................................................................................ 178
7.5.23 getstatic_<t> .............................................................................................. 179
7.5.24 goto ............................................................................................................. 180
7.5.25 goto_w ........................................................................................................ 181
7.5.26 i2b ............................................................................................................... 181
7.5.27 i2s ............................................................................................................... 182
7.5.28 iadd ............................................................................................................. 182
7.5.29 iaload ......................................................................................................... 183
7.5.30 iand ............................................................................................................. 184
7.5.31 iastore ........................................................................................................ 184
7.5.32 icmp ............................................................................................................ 185
7.5.33 icmp ............................................................................................................ 186
7.5.34 idiv ............................................................................................................. 186
7.5.35 if_acmp<cond> ......................................................................................... 187
7.5.36 if_acmp<cond>_w ..................................................................................... 188
7.5.37 if_scmp<cond> ........................................................................................... 189
7.5.38 if_scmp<cond>_w ..................................................................................... 189
7.5.39 if<cond> ................................................................. 190
7.5.40 if<cond>_w ............................................................. 191
7.5.41 ifnonnull .............................................................. 192
7.5.42 ifnonnull_w .......................................................... 193
7.5.43 ifnull ................................................................. 193
7.5.44 ifnull_w .............................................................. 194
7.5.45 iinc ................................................................. 194
7.5.46 iinc_w ............................................................... 195
7.5.47 iloopupswitch ........................................................ 195
7.5.48 iload ................................................................. 196
7.5.49 iload_<n> ........................................................... 197
7.5.50 ilookupswitch ....................................................... 197
7.5.51 imul ................................................................. 198
7.5.52 ineg ................................................................. 199
7.5.53 instanceof .......................................................... 200
7.5.54 invokevirtual ....................................................... 202
  7.5.54.1 Interface Method Resolution .................................. 203
7.5.55 invokespecial ....................................................... 204
7.5.56 invokestatic ........................................................ 205
  7.5.56.1 Super Method Resolution ...................................... 206
7.5.57 invokevirtual ....................................................... 206
  7.5.57.1 Virtual Method Resolution ...................................... 207
7.5.58 ior ................................................................. 208
7.5.59 irem ............................................................... 208
7.5.60 ireturn ............................................................. 209
7.5.61 ishl ............................................................... 210
7.5.62 ishr ............................................................... 210
7.5.63 istore ............................................................... 211
7.5.64 istore_<n> .......................................................... 211
7.5.65 isub ............................................................... 212
7.5.66 itableswitch ........................................................ 213
7.5.67 iushr .............................................................. 214
7.5.68 ixor .................................................................................................................. 214
7.5.69 jsr ...................................................................................................................... 215
7.5.70 new .................................................................................................................... 216
7.5.71 newarray .......................................................................................................... 216
7.5.72 nop .................................................................................................................... 217
7.5.73 pop .................................................................................................................... 218
7.5.74 pop2 .................................................................................................................. 218
7.5.75 putfield_<t> .................................................................................................... 219
7.5.76 putfield_<t>_this ............................................................................................. 220
7.5.77 putfield_<t>_w ............................................................................................... 222
7.5.78 putstatic_<t> .................................................................................................. 223
7.5.79 ret ..................................................................................................................... 224
7.5.80 return ............................................................................................................... 225
7.5.81 s2b ................................................................................................................... 225
7.5.82 s2i .................................................................................................................... 226
7.5.83 sadd .................................................................................................................. 226
7.5.84 saload ............................................................................................................... 227
7.5.85 sand ................................................................................................................. 228
7.5.86 astore ............................................................................................................. 228
7.5.87 sconst_<s> ..................................................................................................... 229
7.5.88 sdiv .................................................................................................................. 229
7.5.89 sinc .................................................................................................................. 230
7.5.90 sinc_w ............................................................................................................. 231
7.5.91 sipush ............................................................................................................. 231
7.5.92 sload ............................................................................................................... 232
7.5.93 sload_<n> ...................................................................................................... 232
7.5.94 slookupswitch ................................................................................................. 233
7.5.95 smul ............................................................................................................... 234
7.5.96 sneg ............................................................................................................... 234
7.5.97 sor ................................................................................................................... 235
7.5.98 srem ............................................................................................................... 235
7.5.99 sreturn ............................................................................................................ 236
7.5.100 sshl ................................................................. 236
7.5.101 sshr ................................................................. 237
7.5.102 sspush ............................................................... 238
7.5.103 sstore ................................................................. 238
7.5.104 sstore_<n> ......................................................... 239
7.5.105 ssub ................................................................. 239
7.5.106 stableswitch ....................................................... 240
7.5.107 sushr ................................................................. 241
7.5.108 swap_x ............................................................... 241
7.5.109 sxor ................................................................. 242

8 Tables of Instructions .................................................. 244

8.1 Instructions by Opcode Value ....................................... 244
8.2 Instructions by Opcode Mnemonic ................................... 248

Glossary .......................................................................... 254

Annex A - Oracle Technology Network Developer License Terms .......... 269
Figures

Figure 1-1: Java Card Application or Library Conversion ................................................................. 23
Figure 1-2: Java Card Application or Library Installation ............................................................... 24
Figure 4-1: Mapping Package Identifiers to AIDs ............................................................................ 54
Figure 4-2: Binary Compatibility Example ......................................................................................... 59
Tables

Table 2-1: Unsupported Java Constant Pool Tags ................................................................. 37
Table 2-2: Supported Java Constant Pool Tags ................................................................. 38
Table 2-3: Support of Java Checked Exceptions ................................................................. 43
Table 2-4: Support of Java Runtime Exceptions .............................................................. 44
Table 2-5: Support of Java Errors ....................................................................................... 44
Table 3-1: Type Support in the Java Card Virtual Machine Instruction Set ......................... 49
Table 3-2: Storage Types and Computational Types .......................................................... 50
Table 4-1: AID Format ....................................................................................................... 53
Table 4-2: Token Range, Type and Scope ......................................................................... 56
Table 4-3: Tokens For Instance Fields .............................................................................. 57
Table 5-1: Export File Constant Pool Tags ....................................................................... 65
Table 5-2: Export File Package Flags .............................................................................. 66
Table 5-3: Export File Class Access and Modifier Flags .................................................. 69
Table 5-4: Export File Field Access and Modifier Flags .................................................. 72
Table 5-5: Export File Method Access and Modifier Flags ................................................ 74
Table 6-1: CAP File Component Tags .............................................................................. 79
Table 6-2: CAP File Component File Names .................................................................... 79
Table 6-3: CAP File Flags ............................................................................................... 83
Table 6-4: CAP File Constant Pool Tags .......................................................................... 94
Table 6-5: Type Descriptor Values .................................................................................. 102
Table 6-6: Encoded Reference Type p1.c1 ...................................................................... 103
Table 6-7: Encoded Byte Array Type .............................................................................. 103
Table 6-8: Encoded Reference Array Type p1.c1 ............................................................ 103
Table 6-9: Encoded Method Signature (JV) ..................................................................... 104
Table 6-10: Encoded Method Signature (Lp1.ci;)S ......................................................... 104
Table 6-11: CAP File Interface and Class Flags ............................................................... 106
Table 6-12: CAP File Method Flags ................................................................................ 122
Table 6-13: Segments of a Static Field Image .................................................................. 124
Table 6-14: Static Field Sizes ......................................................................................... 124
Table 6-15: Array Types ................................................................................................. 126
Table 6-16: One-byte Reference Location Example ....................................................... 129
Table 6-17: CAP File Class Descriptor Flags ................................................................. 136
Table 6-18: CAP File Field Descriptor Flags ................................................................. 138
Table 6-19: Primitive Type Descriptor Values ............................................................... 139
Table 6-20: CAP File Method Descriptor Flags ............................................................... 140
Table 6-21: Class Access and Modifier Flags ................................................................. 146
Table 6-22: Field Access and Modifier Flags .................................................................. 149
Table 6-23: Method Modifier Flags ................................................................................ 151
Preface
Java Card technology combines a subset of the Java programming language with a runtime environment optimized for secure elements, such as smart cards and other tamper-resistant security chips. Java Card technology offers a secure and interoperable execution platform that can store and update multiple applications on a single resource-constrained device, while retaining the highest certification levels and compatibility with standards. Java Card developers can build, test, and deploy applications and services rapidly and securely. This accelerated process reduces development costs, increases product differentiation, and enhances value to customers.

The Classic Edition of the Java Card Platform is defined by three specifications:


This document is a specification of the Classic Edition of the Java Card Platform, Version 3.1, Virtual Machine (Java Card VM).

In this book, Java Card Platform refers to version 3.1 to distinguish it from all earlier versions. A vendor of a Java Card technology-enabled device provides an implementation of the Java Card RE. An implementation within the context of this specification refers to a vendor's implementation of the Java Card Virtual Machine (or Java Card VM), the Java Card Application Programming Interface (API), or other component, based on the Java Card technology specifications. A "reference implementation" is an implementation produced by Oracle. Application software written for the Java Card platform is referred to as a Java Card technology-based applet (Java Card applet or card applet).

Who Should Use This Specification
This specification is intended to assist implementers of the Java Card RE in creating an implementation, developing a specification to extend the Java Card technology specifications, or in creating an extension to the runtime environment for the Java Card platform. This specification is also intended for Java Card applet developers who want a greater understanding of the Java Card technology specifications.

Before You Read This Specification
Before reading this guide, you should be familiar with the Java programming language, the other Java Card technology specifications, and smart card technology. A good resource for becoming familiar with Java technology and Java Card technology located at:
http://www.oracle.com/technetwork/java/javacard/overview/

Shell Prompts

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell</td>
<td><code>machine-name%</code></td>
</tr>
<tr>
<td>C shell superuser</td>
<td><code>machine-name#</code></td>
</tr>
<tr>
<td>Bourne shell and Korn shell</td>
<td><code>§</code></td>
</tr>
<tr>
<td>Bourne shell and Korn shell superuser</td>
<td><code>#</code></td>
</tr>
</tbody>
</table>
## Typographic Conventions

The settings on your browser might differ from these settings.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. % You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, when contrasted with on-screen computer output</td>
<td><code>%su</code> &lt;br&gt;<code>Password:</code></td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.</td>
<td>Read Chapter 6 in the <em>User's Guide</em>. &lt;br&gt;These are called <code>class</code> options. &lt;br&gt;You <em>must</em> be superuser to do this. &lt;br&gt;To delete a file, type <code>rm filename</code>.</td>
</tr>
</tbody>
</table>

## Related Documentation

References to various documents or products are made in this guide, so you might want to have them available:

- *Application Programming Interface, Java Card Platform, Version 3.1, Classic Edition*

## Third-Party Web Sites

Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.
**Documentation Accessibility**
For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at:


**Access to Oracle Support**
Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit:


Or, if you are hearing impaired, visit:


**Oracle Welcomes Your Comments**
Oracle is interested in improving its documentation and welcomes your comments and suggestions.

Please include the title of your document with your feedback:

1

Introduction

This document specifies the Java Card Virtual Machine features required by the Classic Edition of Java Card technology.

- It defines the subset of the Java Virtual Machine used for the Java Card Virtual Machine and list the supported and unsupported features.
- It defines the binary representation of the application code, the role and structure of the Export and CAP file formats and their use in the verification and linking process.
- It specifies the Java Card Virtual Machine byte-code set and its detailed behavior.

1.1 Motivation

Java Card technology enables programs written in the Java programming language to be run on secure elements such as smart cards and other tamper-resistant security chips. 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 (to distinguish it from browser applets).

While Java Card technology enables programs written in the Java programming language to run on small devices such as smart cards, those 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 Java Virtual Machine Specification, smart card and secure elements manufacturers could construct their own Java Card technology-based implementations (“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 embedded security and smart card industry. In a world with billions of secure elements with varying processors and configurations, the costs of supporting multiple binary formats for software distribution could be overwhelming.

This Virtual Machine Specification, Java Card Platform, v3.1, Classic Edition 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 Virtual Machine Specification, Java Card Platform, v3.1, Classic Edition 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 deployed to be 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 software for the Java Card platform. There are several components that make up a Java Card system, including the Java Card virtual machine, the Converter for the Java Card platform (“Java Card Converter”), a terminal installation tool, and an installation program that runs on the device, as shown in Figure 1-1 and Figure 1-2.

Figure 1-1: Java Card Application or Library Conversion
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 all of the class files in one or more Java packages which make up a Java Card CAP file. A Java package that contains one or more non-abstract subclasses, direct or indirect, of the javacard.framework.Applet class is referred to as an applet package. Otherwise the package is referred to as a library package. A Java Card CAP file may contain only applet packages, only library packages or a combination of applet and library packages. Additionally, both applet and library packages in a Java Card CAP file can be public or private.

A private library package in a Java Card CAP file is not listed in the Export Component (6.13 Export Component) of the CAP file and is therefore not visible outside the Java Card CAP file. Similarly, a private applet package in a Java Card CAP file is not listed in the Export Component (6.13 Export Component) of the CAP file, however, non-abstract direct or indirect subclasses of the javacard.framework.Applet class are listed in the Applet Component (6.6 Applet Component) of the CAP file. For a public applet package in a Java Card CAP file, only public interfaces extending javacard.framework.Shareable are listed in the Export Component (6.13 Export Component) of the CAP file and are therefore visible outside the Java Card CAP file. For further details see The CAP File Format.

The Java Card Converter also takes as input 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. The converter can also produce export files for the public applet and library packages in a CAP file.

After conversion, the CAP file is copied to a 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 needs not load or manipulate CAP files; it only needs to 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 instructions for the Java Card platform (“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.

The Java Card virtual machine specification mandates CAP file verification. The data needed for verification is packaged separately from the data needed for the actual execution of the code (see 6.14 Descriptor Component). This enables the following two options for the implementation of the verification, depending on device capabilities, resources and deployment model:

- Perform CAP file verification inside the device (on-device verification),
- Perform CAP file verification outside the device (off-device verification).

The CAP file verification must be performed at least once, before loading, before installation or before execution of the code, in order to ensure that each bytecode is valid at execution time. Off-device verification can also be systematically used to detect any issue before initiating deployment.

When off-device verification option is used, the deployment process must also ensure that:

- Verification uses export files that are binary compatible with API packages installed on the device (see 5 The Export File Format).
- After verification, the CAP file to be executed on the device is not altered in a way that does not satisfy the constraints checked by this verification.

1.4 Java Card Runtime Environment Security
The standard runtime environment for the Java Card platform is the Java Card Runtime Environment (Java Card RE). The Java Card RE 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 Java Card RE imposes additional runtime security requirements on devices that implement the Java Card RE, which results in a need for additional features on the Java Card virtual machine. Throughout this document, these additional features are designated as Java Card RE-specific.
The basic runtime security feature imposed by the Java Card RE 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 Java Card RE 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 shareable interface objects, see the description of the interface javacard.framework.Shareable in the *Application Programming Interface, Java Card Platform, v3.1, Classic Edition*. Some descriptions of firewall-related features make reference to the Shareable interface.

The applet firewall also protects from unauthorized use the objects owned by the Java Card RE itself. The Java Card RE can use mechanisms not reflected in the Java Card API to make its objects available for use by applets. A full description of the Java Card RE-related isolation and sharing features can be found in the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*. 
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 platform.

2.1 Why a Subset is Needed
It would be ideal if programs for secure elements such as 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 1.2K of RAM, 16K of non-volatile memory (EEPROM or flash) and 32K-48K 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 Platform 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.

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
The following features are not supported.

2.2.1.1.1 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 device during manufacturing or downloaded through an installation process after it has been issued. Programs
executing on the device may only refer to classes that already exist on the device, since there is no way to download classes during the normal execution of application code.

2.2.1.1.2 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.

2.2.1.1.3 Finalization
Finalization is also not supported. finalize() will not be called automatically by the Java Card virtual machine.

2.2.1.1.4 Threads
The Java Card virtual machine does not support multiple threads of control. Programs for the Java Card platform (“Java Card programs”) cannot use class Thread or any of the thread-related keywords in the Java programming language.

2.2.1.1.5 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.

2.2.1.1.6 Access Control in Java Packages
The Java Card platform language subset supports the package access control defined in the Java language. However, the cases that are not supported are as follows.

- 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.
- A public class cannot contain a public or protected field of type reference to a package-visible class.
- A public class cannot contain a public or protected method with a return type of type reference to a package-visible class.
- A public or protected method in a public class cannot contain a formal parameter of type reference to a package-visible class.
- A package-visible class that is extended by a public class cannot define any public or protected methods or fields.
- A package-visible interface that is implemented by a public class cannot define any fields.
- A package-visible interface cannot be extended by an interface with public access visibility.

2.2.1.1.7 Typesafe Enums
The Java Card platform language subset does not support the enumerated type facility and the keyword enum.
2.2.1.1.8 Enhanced for Loop
The Java Card platform language subset does not support the enhanced for loop language construct. Support for the enhanced for loop construct requires support for array indexing using the integer data type. The Java Card platform only supports array indexing using the short data type.

2.2.1.1.9 Varargs
The Java Card platform language subset does not support variable-length argument lists. The variable-length argument construct requires the compiler to generate code that creates a new array object each time a variable-length argument array method is invoked, thereby causing implicit memory allocations in Java Card runtime memory heap.

2.2.1.1.10 Runtime Visible Metadata (Annotations)
The Java Card platform does not support this language feature which lets you introduce meta-data information into the runtime environment to be accessed reflectively. The Java Card platform does not support reflection.

2.2.1.1.11 Assertions
The Java Card runtime does not provide runtime support for statements in the Java programming language called assertions that are used to test assumptions about program functionality.

2.2.1.2 Unsupported Keywords
The following keywords indicate unsupported options related to native methods, threads, floating point, memory management, and debugging:

- native
- strictfp
- synchronized
- enum
- transient
- assert
- volatile

2.2.1.3 Unsupported Types
The Java Card platform does not support types:

- char
- double
- float
- long
- arrays of more than one dimension.

2.2.1.4 Unsupported Classes
In general, none of the Java programming language core API classes are supported in the Java Card platform. Some classes from the java.lang package are supported (see Section 2.2.2.4 Supported
Classes), 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.

2.2.1.4.1 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 Supported Features
The following features are the more important supported features.

2.2.2.1.1 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 Unsupported Features), packages in the Java Card platform are used exactly the way they are in the Java platform.

2.2.2.1.2 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.

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. See 2.2.3.2 Object Deletion Mechanism for more information regarding support for an optional object deletion mechanism.

2.2.2.1.3 Virtual Methods
Since Java Card technology-based objects (“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.

2.2.2.1.4 Interfaces
Java Card API 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.

2.2.2.1.5 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 Exceptions for specification of errors and exceptions.

2.2.2.1.6 Generics
This Java language facility allows a type or method to operate on objects of various types while providing compile-time type safety. It adds compile-time type safety and eliminates the need for casting.

2.2.2.1.7 Static Import
This Java language facility lets you avoid importing an entire class simply to access its static members or qualifying static members with class names each time it is used.

2.2.2.1.8 Runtime Invisible Metadata (Annotations)
This language feature lets you avoid writing boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. The Java Card platform language subset supports the use of annotations which are not visible at runtime. These annotations do not themselves use the runtime visible meta-data annotation @Retention(RetentionPolicy.RUNTIME).

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

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

2.2.2.3 Supported Types
Java programming language types supported:

• boolean
• byte
• short
• int
• Objects (class instances and single-dimensional arrays)
• 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 Integer Data Type).

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

2.2.2.4.1 Object
Java Card API 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.

2.2.2.4.2 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 platform-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 Integer Data Type
The int keyword and 32-bit integer data type 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.

The result of an arithmetic expression produced by a Java Card virtual machine must be equal to the result produced by a Java virtual machine, regardless of the input values. A Java Card virtual machine that does not support the int data type must reject expressions that could produce a different result.

2.2.3.2 Object Deletion Mechanism
The Java Card platform offers an optional, object deletion mechanism. Applications designed to run on these implementations can use the facility by invoking the appropriate API. See Application Programming Interface, Java Card Platform, v3.1, Classic Edition. But, the facility is best suited for updating large objects such as certificates and keys atomically. Therefore, application programmers should conserve on the allocation of objects.

2.2.4 Limitations of the Java Card Virtual Machine
The limitations of resource-constrained hardware prevent Java Card virtual machines 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. Java packages included in a Java Card CAP file that do not violate these maximum values can be converted into Java Card technology-based CAP files (“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 Limitations of Packages
The following are limitations of packages.

2.2.4.1.1 Packages in a Java Card CAP file
A Java Card CAP file can contain at most 255 packages.

2.2.4.1.2 Package References
A package can reference at most 128 other packages external to the Java Card CAP file containing the package.

2.2.4.1.3 Package Name
The fully qualified name of a package may contain a maximum of 255 characters. The package name size is further limited if it contains one or more characters which, when represented in UTF-8 format, requires multiple bytes.
2.2.4.2 Limitations of Classes
The following are limitations of classes.

2.2.4.2.1 Classes in a Package
A package can contain at most 255 classes and interfaces.

2.2.4.2.2 Interfaces
A class can implement at most 15 interfaces, including interfaces implemented by super classes.

An interface can inherit from at most 14 super interfaces.

2.2.4.2.3 Static Fields
A class in an applet package can have at most 256 public or protected static non-final fields. A class in a library package can have at most 255 public or protected static non-final fields. There is no limit on the number of static final fields (constants) declared in a class.

2.2.4.2.4 Static Methods
A class in an applet package can have at most 256 public or protected static methods. A class in a library package can have at most 255 public or protected static methods.

2.2.4.3 Limitations of Objects
The following are limitations of objects.

2.2.4.3.1 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.

2.2.4.3.2 Class Instances
Class instances can contain a maximum of 255 fields, where an int data type is counted as occupying two fields. These limits include inherited fields.

2.2.4.3.3 Arrays
Arrays can hold a maximum of 32767 components.

2.2.4.4 Limitations of Methods
The maximum number of variables that can be used in a method is 255. This limit includes local variables, method parameters, and, in the case of an instance method invocation, a reference to the object on which the instance method is being invoked (meaning, this). 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 technology-based bytecodes (“Java Card bytecodes”) may differ from the number of Java bytecodes in the Java virtual machine implementation of that method.

The maximum depth of an operand stack associated with a method is 255 16-bit cells.
2.2.4.5 Limitations of 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 (2.2.4.4 Limitations of Methods).

2.2.4.6 Limitations of Class Initialization
The Java Card virtual machine contains limited support for class initialization because there is no general mechanism for executing <clinit> methods. Support for <clinit> methods is limited to the initialization of static field values with the following constraints:

- Static fields of applet CAP files may only be initialized to primitive compile-time constant values, or arrays of primitive compile-time constants.
- Static fields in interfaces must only be initialized to primitive compile-time constant.
- Static fields of CAP files containing only user libraries may only be initialized to primitive compile-time constant values.
- Only static fields declared in the current class may be initialized in the <clinit> method.

Primitive constant data types include boolean, byte, short, and int.

Given Java technology source files that adhere to these language-level constraints on static field initialization, it is expected that reasonable Java compilers will:

- Inline constants in the bytecodes that reference static final primitive fields that are initialized in the declaration statement.
- Produce only the following bytecodes:
  - load a value on the stack: iconst_[-m1,0-5], [b|s]ipush, ldc, ldc_w, aconst_null
  - create an array: newarray([byte|short|boolean|int])
  - duplicate items on the stack: dup
  - store values in arrays or static fields: [b|i|s]astore, putstatic
  - return from method: return

2.2.5 Multiselectable Applets Restrictions
Applets that implement the javacard.framework.Multiselectable interface are called multiselectable applets. For more details on multiselection, please see the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.

All applets within a CAP file shall be multiselectable, or none shall be.

2.2.6 Java Card Platform Remote Method Invocation (RMI) Restrictions
This section defines the subset of the RMI system that is supported by Java Card platform RMI ("Java Card RMI").
2.2.6.1 Remote Classes and Remote Interfaces
A class is remote if it or any of its superclasses implements a remote interface. A remote interface is an interface which satisfies the following requirements:

- The interface name is `java.rmi.Remote` or the interface extends, directly or indirectly, the interface `java.rmi.Remote`.
- Each method declaration in the remote interface or its super-interfaces includes the exception `java.rmi.RemoteException` (or one of its superclasses) in its `throws` clause.
- In a remote method declaration, if a remote object is declared as a return type, it is declared as the remote interface, not the implementation class of that interface.

In addition, Java Card RMI imposes additional constraints on the definition of remote methods. These constraints are as a result of the Java Card platform language subset and other feature limitations. For more information, see 2.2.6.2 Access Control of Remote Interfaces and 2.2.6.3 Parameters and Return Values.

2.2.6.2 Access Control of Remote Interfaces
The Java RMI system supports the package access control defined in the Java language. However, Java Card RMI does not support package-visible remote interfaces.

2.2.6.3 Parameters and Return Values
The parameters of a remote method must only include parameters of the following types:

- Any primitive type supported by Java Card technology (`boolean, byte, short, int`)
- Any single-dimension array type of a primitive type supported by Java Card technology (`boolean[], byte[], short[], int[]`)

The return type of a remote method must only be one of the following types:

- Any primitive type supported by Java Card technology (`boolean, byte, short, int`)
- Any single-dimension array type of a primitive type supported by Java Card technology (`boolean[], byte[], short[], int[]`)
- Any remote interface type
- Type `void`

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 *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
The following items are not supported in class files.

2.3.1.1.1 Class Access Flags
In class_info and interface_info structures, the access flag ACC_ENUM is not supported.

2.3.1.1.2 Field Descriptors
Field descriptors may not contain BaseType characters C, D, F or J. ArrayType descriptors for arrays of more than one dimension may not be used.

2.3.1.1.3 Constant Pool
Constant pool table entries with the following tag values are not supported.

Table 2-1: Unsupported Java Constant Pool Tags

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT_String</td>
<td>8</td>
</tr>
<tr>
<td>CONSTANT_Float</td>
<td>4</td>
</tr>
<tr>
<td>CONSTANT_Long</td>
<td>5</td>
</tr>
<tr>
<td>CONSTANT_Double</td>
<td>6</td>
</tr>
</tbody>
</table>

2.3.1.1.4 Fields
In field_info structures, the access flags ACC_VOLATILE, ACC_TRANSIENT and ACC_ENUM are not supported.

2.3.1.1.5 Methods
In method_info structures, the access flags ACC_SYNCHRONIZED, ACC_STRICT, ACC_NATIVE, and ACC_VARARGS are not supported.

2.3.1.2 Supported in Class Files
The following items are supported in class files.

2.3.1.2.1 ClassFile
All items in the ClassFile structure are supported.

2.3.1.2.2 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.
2.3.1.2.3 Method Descriptors
All forms of method descriptors are supported.

2.3.1.2.4 Constant Pool
Constant pool table entry with the following tag values are supported.

Table 2-2: Supported Java Constant Pool Tags

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT_Class</td>
<td>7</td>
</tr>
<tr>
<td>CONSTANT_Fieldref</td>
<td>9</td>
</tr>
<tr>
<td>CONSTANT_Methodref</td>
<td>10</td>
</tr>
<tr>
<td>CONSTANT_InterfaceMethodref</td>
<td>11</td>
</tr>
<tr>
<td>CONSTANT_Integer</td>
<td>3</td>
</tr>
<tr>
<td>CONSTANT_NameAndType</td>
<td>12</td>
</tr>
<tr>
<td>CONSTANT_Utf8</td>
<td>1</td>
</tr>
</tbody>
</table>

2.3.1.2.5 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.

2.3.1.2.6 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.

2.3.1.2.7 Attributes
The attribute_info structure is supported. The Code, ConstantValue, Exceptions, LocalVariableTable, Synthetic, InnerClasses, RuntimeInvisibleAnnotations, RuntimeInvisibleParameterAnnotations and Deprecated 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 7, Java Card Virtual Machine Instruction Set.

2.3.2.1 Unsupported Bytecodes
The unsupported bytecodes are:

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

2.3.2.2 Supported Bytecodes

The supported bytecodes are:

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

2.3.2.3 Static Restrictions on Bytecodes
A class file must conform to the following restrictions on the static form of bytecodes.

2.3.2.3.1 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 Integer Data Type).

2.3.2.3.2 lookupswitch
The value of the npairs operand must be less than 65536. This limit is far greater than the limit imposed by the maximum size of methods (2.2.4.4 Limitations of Methods). 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 Integer Data Type). Otherwise, key values must be in the range -32768 to 32767.

2.3.2.3.3 tableswitch
The bytecode can contain at most 65536 cases. This limit is far greater than the limit imposed by the maximum size of methods (2.2.4.4 Limitations of Methods). If a program does not use the optional int instruction (2.2.3.1 Integer Data Type), the values of the high and low operands must both be at least -32768 and at most 32767.

2.3.2.3.4 wide
The wide bytecode can only be used with an iinc instruction.
2.3.3 Exceptions

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

Not all of the Java platform’s exceptions are supported in the Java Card platform. Exceptions related to unsupported features are naturally not supported. Class loader exceptions (the bulk of the checked exceptions) are not 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

Support of Java checked exceptions:

Table 2-3: Support of Java Checked Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Supported or Not Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassNotFoundException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>CloneNotSupportedException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>IllegalAccessException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>InstantiationException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>InterruptedException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>NoSuchFieldException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>NoSuchMethodException</td>
<td>Not Supported</td>
</tr>
</tbody>
</table>
### 2.3.3.3 Runtime Exceptions

Support of Java Runtime Exceptions:

**Table 2-4: Support of Java Runtime Exceptions**

<table>
<thead>
<tr>
<th>Runtime Exception</th>
<th>Supported or Not Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticException</td>
<td>Supported</td>
</tr>
<tr>
<td>ArrayStoreException</td>
<td>Supported</td>
</tr>
<tr>
<td>ClassCastException</td>
<td>Supported</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>IllegalThreadStateException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>NumberFormatException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>IllegalMonitorStateException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>IllegalStateException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>IndexOutOfBoundsException</td>
<td>Supported</td>
</tr>
<tr>
<td>ArrayIndexOutOfBoundsException</td>
<td>Supported</td>
</tr>
<tr>
<td>StringIndexOutOfBoundsException</td>
<td>Not Supported</td>
</tr>
<tr>
<td>NegativeArraySizeException</td>
<td>Supported</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>Supported</td>
</tr>
<tr>
<td>SecurityException</td>
<td>Supported</td>
</tr>
</tbody>
</table>

### 2.3.3.4 Errors

Support of Java errors:

**Table 2-5: Support of Java Errors**

<table>
<thead>
<tr>
<th>Error</th>
<th>Supported or Not Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssertionError</td>
<td>Not Supported</td>
</tr>
<tr>
<td>LinkageError</td>
<td>Supported</td>
</tr>
<tr>
<td>ClassCircularityError</td>
<td>Supported</td>
</tr>
<tr>
<td>ClassFormatError</td>
<td>Supported</td>
</tr>
<tr>
<td>ExceptionInInitializerError</td>
<td>Supported</td>
</tr>
<tr>
<td>IncompatibleClassChangeError</td>
<td>Supported</td>
</tr>
<tr>
<td>AbstractMethodError</td>
<td>Supported</td>
</tr>
<tr>
<td>IllegalAccessException</td>
<td>Supported</td>
</tr>
<tr>
<td>InstantiationException</td>
<td>Supported</td>
</tr>
<tr>
<td>NoSuchFieldError</td>
<td>Supported</td>
</tr>
<tr>
<td>NoSuchMethodError</td>
<td>Supported</td>
</tr>
<tr>
<td>NoClassDefFoundError</td>
<td>Supported</td>
</tr>
<tr>
<td>UnsatisfiedLinkError</td>
<td>Supported</td>
</tr>
<tr>
<td>VerifyError</td>
<td>Supported</td>
</tr>
<tr>
<td>ThreadDeath</td>
<td>Not Supported</td>
</tr>
<tr>
<td>VirtualMachineError</td>
<td>Supported</td>
</tr>
<tr>
<td>InternalError</td>
<td>Supported</td>
</tr>
<tr>
<td>OutOfMemoryError</td>
<td>Supported</td>
</tr>
<tr>
<td>Error</td>
<td>Supported or Not Supported</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>StackOverflowError</td>
<td>Supported</td>
</tr>
<tr>
<td>UnknownError</td>
<td>Supported</td>
</tr>
<tr>
<td>UnsupportedClassVersionError</td>
<td>Supported</td>
</tr>
</tbody>
</table>
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, Second 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*, the *boolean type*, and the *returnAddress type*. The numeric types consist only of these 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 the *boolean type* is identical to that in the Java virtual machine. The value 1 is used to represent true and the value of 0 is used to represent false.

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 CAP files in which applets are defined. An easy way to think of a context is as the runtime equivalent of a CAP file. As a consequence, all applet instances from the same CAP file will share the same context.

The Java Card Runtime Environment also has its own context. Framework objects execute in this Java Card RE 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 2.2.4.6 Limitations of Class Initialization.

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 programs.

The CAP file format describes files that contain executable code and can be downloaded and installed onto a Java Card technology-enabled device. A CAP file is produced by a Java Card Platform Converter tool, and contains a converted form of one or more entire packages 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 API 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. 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 NONE, 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.

Table 3-1: Type Support in the Java Card Virtual Machine Instruction Set

<table>
<thead>
<tr>
<th>opcode</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tspush</td>
<td>bspush</td>
<td>sspush</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>Tipush</td>
<td>bipush</td>
<td>sipush</td>
<td>iipush</td>
<td>NONE</td>
</tr>
<tr>
<td>Tconst</td>
<td>NONE</td>
<td>sconst</td>
<td>iconst</td>
<td>aconst</td>
</tr>
<tr>
<td>Tload</td>
<td>NONE</td>
<td>sload</td>
<td>iload</td>
<td>aaload</td>
</tr>
<tr>
<td>Tstore</td>
<td>NONE</td>
<td>sstore</td>
<td>istore</td>
<td>astore</td>
</tr>
<tr>
<td>Tinc</td>
<td>NONE</td>
<td>sinc</td>
<td>iinc</td>
<td>NONE</td>
</tr>
<tr>
<td>Taload</td>
<td>baload</td>
<td>saload</td>
<td>iaload</td>
<td>aaload</td>
</tr>
<tr>
<td>Tastore</td>
<td>bastore</td>
<td>sastore</td>
<td>iastore</td>
<td>aastore</td>
</tr>
<tr>
<td>Tadd</td>
<td>NONE</td>
<td>sadd</td>
<td>iadd</td>
<td>NONE</td>
</tr>
<tr>
<td>Tsub</td>
<td>NONE</td>
<td>ssub</td>
<td>isub</td>
<td>NONE</td>
</tr>
<tr>
<td>Tmul</td>
<td>NONE</td>
<td>smul</td>
<td>imul</td>
<td>NONE</td>
</tr>
<tr>
<td>Tdiv</td>
<td>NONE</td>
<td>sdiv</td>
<td>idiv</td>
<td>NONE</td>
</tr>
<tr>
<td>Trem</td>
<td>NONE</td>
<td>srem</td>
<td>irem</td>
<td>NONE</td>
</tr>
<tr>
<td>Tneg</td>
<td>NONE</td>
<td>sneg</td>
<td>ineg</td>
<td>NONE</td>
</tr>
<tr>
<td>Tshl</td>
<td>NONE</td>
<td>sshl</td>
<td>ilshl</td>
<td>NONE</td>
</tr>
<tr>
<td>Tshr</td>
<td>NONE</td>
<td>ssrr</td>
<td>ishr</td>
<td>NONE</td>
</tr>
<tr>
<td>Tushr</td>
<td>NONE</td>
<td>sushr</td>
<td>iushr</td>
<td>NONE</td>
</tr>
</tbody>
</table>
The mapping between Java storage types and Java Card virtual machine computational types is summarized in Table 3-2.

### Table 3-2: Storage Types and Computational Types

<table>
<thead>
<tr>
<th>Java (Storage) Type</th>
<th>Size in Bits</th>
<th>Computational Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8</td>
<td>short</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>short</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>int</td>
</tr>
</tbody>
</table>

Chapter 7 describes the Java Card virtual machine instruction set in detail.
4 Binary Representation

This chapter presents information about the binary representation of Java Card programs. Java Card technology-based binaries ("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 Java Archive (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 API packages. The fourth and fifth sections describe the constraints for upward compatibility between different versions of a Java Card technology-based binary ("Java Card binary") program file, and versions assigned based upon that compatibility.

4.1 Java Card Platform 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 technology-based export file ("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, The Export File Format.

4.1.2 CAP File Format

A Java Card CAP file contains a binary representation of a Java Card application or library or both, consisting of one or more packages of classes that can be installed on a device and used to execute the Java Card application or library’s classes on a Java Card virtual machine.

A CAP file is produced by a Java Card converter when a Java Card application or library is converted. 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). A CAP file can be in Compact or Extended format where a CAP file in Compact format can only contain a single Java package and a CAP file in Extended format may contain one or more packages.

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

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, The CAP File Format).

CAP file components are stored as files in a JAR file. Each CAP file component is located in a directory called javacard. In CAP files in Compact format javacard subdirectory is in a directory representing the package. For example, the CAP file components of the package com.oracle.framework are located in the directory com/oracle/framework/javacard. In CAP files in Extended format, javacard subdirectory is in a directory representing the CAP file. For example, for a CAP file called com.oracle.helloworld that may contain multiple packages, the CAP file components of the CAP file are located in directory com/oracle/helloworld/javacard.

Export files may also be contained in a JAR file, whether that JAR file contains CAP file components or not. If an export component is included in the CAP file in Compact format, it must be located in the same directory as the components for that package would be. If export files are included in the CAP file in Extended format, they must be located in the directory javacard which is a subdirectory representing each package which is represented by each export file. For example, the export file for package com.oracle.framework is located in directory com/oracle/framework/javacard.

The name of a JAR file containing CAP file components is not defined as part of this specification. Other files, including CAP file components for another package, may also reside in a JAR file that contains CAP file components.
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, packages and CAP files. 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 Table 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.

<table>
<thead>
<tr>
<th>Resource Identifier</th>
<th>Proprietary Identifier Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID (5 bytes)</td>
<td>PIX (0-11 bytes)</td>
</tr>
</tbody>
</table>

ISO controls the assignment of RIDs to companies, with each company obtaining its own unique RID from the ISO. Companies manage assignment of PIIXs 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, CAP files, packages and applets are identified using AIDs.

4.2.2.1 CAP File AID namespace

All CAP files must be assigned an AID such that no two CAP files have the same AID. The AID for a CAP file is constructed from the concatenation of the company’s RID and a PIX for that CAP file.

4.2.2.2 Applet AID namespace

Each applet loaded on a Java Card technology enabled device must have an AID. This AID is constructed similarly to a CAP file 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 other applet of the same CAP file. The RID of each applet in a CAP file must be the same as the RID of the CAP file AID.
4.2.2.3 Package AID namespace

Any package that is represented in an export file must be assigned an AID such that no two packages have the same 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 4-1.

Figure 4-1: Mapping Package Identifiers to AIDs

4.2.2.3 Custom Component AID namespace

Custom components defined in a CAP file are also identified using AIDs. Like AIDs for applets, packages and CAP files, custom component AIDs are formed by concatenating the RID and a PIX.

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.

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 three kinds of items that can be assigned private tokens.

• Instance Fields
• Virtual Methods
• Packages

4.3.3 The Export File and Conversion
An export file contains entries for externally visible items in the package. 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 CAP’s constant pool. References to items in other CAP files 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 owner and published in the package export file(s) for that API. Since the name-to-token mappings are published, an API owner 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 4-2.

Table 4-2: Token Range, Type and Scope

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Range</th>
<th>Type</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>0 - 127</td>
<td>Private</td>
<td>Package</td>
</tr>
<tr>
<td>Class</td>
<td>0 - 254</td>
<td>Public</td>
<td>Package</td>
</tr>
<tr>
<td>Static Field</td>
<td>0 - 255</td>
<td>Public</td>
<td>Class</td>
</tr>
<tr>
<td>Static Method</td>
<td>0 - 255</td>
<td>Public</td>
<td>Class</td>
</tr>
<tr>
<td>Instance Field</td>
<td>0 - 255</td>
<td>Public or Private</td>
<td>Class</td>
</tr>
<tr>
<td>Virtual Method</td>
<td>0 - 127</td>
<td>Public or Private</td>
<td>Class Hierarchy</td>
</tr>
<tr>
<td>Interface Method</td>
<td>0 - 127</td>
<td>Public</td>
<td>Class</td>
</tr>
</tbody>
</table>

4.3.7.1 Package

All package references from within a CAP file are assigned private package tokens. 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 and interfaces in a package are assigned public class tokens. Class token values must be in the range from 0 to 254, inclusive. The tokens for all the public classes and interfaces in a package are numbered consecutively starting at zero. The ordering of class tokens is not specified.

Package-visible classes and interfaces are not assigned tokens.
### 4.3.7.3 Static Fields
All externally visible static fields in a package are assigned public static field tokens. The tokens for all 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.

Package-visible and private static fields are not assigned tokens. In addition, no tokens are assigned for final static fields that are initialized to primitive, compile-time constants, as these fields are never represented as fields in CAP files.

### 4.3.7.4 Static Methods and Constructors
All externally visible static methods and constructors in a package are assigned public static method tokens. Constructors are included in this category because they are statically bound. Static method token values must be in the range from 0 to 255, inclusive. The tokens for all the externally visible static methods and constructors in a class are numbered consecutively starting at zero. The ordering of static method tokens is not specified.

Package-visible and private static methods as well as package-visible and private constructors are not assigned tokens.

### 4.3.7.5 Instance Fields
All instance fields defined in a package are assigned either public or private instance field tokens. The scope of a set of instance field tokens is limited to the class that declares the instance fields, not including the fields declared by superclasses of that class.

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.

Within a class, 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.

<table>
<thead>
<tr>
<th>Visibility</th>
<th>Category</th>
<th>Type</th>
<th>Token Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>public and protected fields (public tokens)</td>
<td>primitive</td>
<td>boolean</td>
<td>0</td>
</tr>
<tr>
<td>public and protected fields (public tokens)</td>
<td>primitive</td>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>public and protected fields (public tokens)</td>
<td>primitive</td>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>public and protected fields (public tokens)</td>
<td>reference</td>
<td>byte[]</td>
<td>3</td>
</tr>
<tr>
<td>public and protected fields (public tokens)</td>
<td>reference</td>
<td>Applet</td>
<td>4</td>
</tr>
<tr>
<td>package and private fields (private tokens)</td>
<td>reference</td>
<td>short[]</td>
<td>5</td>
</tr>
<tr>
<td>package and private fields (private tokens)</td>
<td>reference</td>
<td>Object</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 4-3: Tokens For Instance Fields
<table>
<thead>
<tr>
<th>Visibility</th>
<th>Category</th>
<th>Type</th>
<th>Token Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>package and private fields (private tokens)</td>
<td>primitive</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>package and private fields (private tokens)</td>
<td>primitive</td>
<td>short</td>
<td>9</td>
</tr>
</tbody>
</table>

### 4.3.7.6 Virtual Methods

Virtual methods are instance methods that are resolved dynamically. The set includes all public, protected and package-visible instance methods. Private instance methods and all constructors are not virtual methods, but instead are resolved statically during compilation.

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.

If a method overrides an externally visible (public or protected) method implemented in the class’s superclass, that method is assigned 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 to package-visible virtual methods. They 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 method token 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. Each interface includes its own token values for all the methods inherited from super-interfaces as well as its defined methods. 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 classes are grouped by package into a CAP file, therefore the granularity of binary compatibility is between packages. The binary representation of classes is represented by the components in a CAP file, and the API of every single package is represented in an export file.

In a Java Card system, a change to a type in a Java package in a Java Card CAP file 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 files of packages included in the preexisting CAP file can link with the new CAP file without errors.

Figure 4-2 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.

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.
Tokens are used to resolve references to imported elements of a package. Tokens assigned to public and protected virtual methods are scoped to a class. 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.

### 4.5 CAP and Package Versions

Each CAP file in the Java Card system is assigned a unique CAP file version number. Furthermore, each implementation of a package in a Java Card system is assigned a pair of major and minor version numbers. Version numbers for a package 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 CAP file and contained packages are assigned by the CAP file and package provider. It is recommended that the initial implementation of a CAP file and 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.

For a package, major version must be changed when a new implementation 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.

Rules for specifying the version numbers for the CAP file in Extended Format (The CAP File Format) are beyond the scope of this specification. For the CAP file in Compact Format (The CAP File Format) the CAP file version number must be the same as the version number of the single package that it contains.

#### 4.5.2 Linking

Both an export file and a CAP file contain the major and minor version numbers of the packages described. When a CAP file is installed on a Java Card technology-enabled device, resident images of the packages it contains are created and the major and minor version numbers are recorded as part of that
images. 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 packages 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.
The Export File Format

This chapter describes the 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.

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 \texttt{u1}, \texttt{u2}, and \texttt{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 \textit{items}. Unlike the fields of a C structure, successive items are stored in the Java Card platform file sequentially, without padding or alignment.

Variable-sized \textit{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 \textit{array}, the elements are equal in size.

5.1 Export File Name
As described in 4.1.1 Export File Format, the name of an 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 JAR File Container, 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 of the corresponding package’s directory. For example, the framework.exp file would be located in the subdirectory javacard/framework/javacard.

5.3 Ownership
An export file is owned by the entity that owns the package it represents. The owner of a package defines the API of that package and may or may not provide all implementations of that package.
implementations, however, must conform to the definition provided in the export file provided by the owner.

A particular example of export file ownership is the Java Card API packages. Oracle defines these packages. Oracle also provides the export files for these packages. All implementations of the Java Card API packages must conform to the definitions provided by Oracle and comply with the token assignments provided in these export files.

5.4 Hierarchies Represented
Classes and interfaces represented in an export file include public elements defined within their respective hierarchies. For example, instead of indicating the immediate superclass or superinterface, all public superclasses or superinterfaces are listed. This design concept is applied not only to superclasses or superinterfaces, but also to virtual methods and implemented interfaces.

5.5 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 referenced_package_count (since Export File format 2.3)
    u2 referenced_packages[referenced_package_count] (since Export File format 2.3)
    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. Together, a major and a minor version number determine the version of the export file format. If an export file has the major version number of M and minor version number of m, the version of the export file’s format is M.m.
A change in the major version number indicates a major incompatibility change, one that requires a fundamentally different Java Card virtual machine. A Java Card virtual machine is not required to support export files with different major version numbers. A Java Card virtual machine is required to support export files having a given major version number and all valid minor version numbers in the range 0 through some particular \texttt{minor\_version} where a valid minor version number is a minor version number that has been defined in a version of the Java Card virtual machine specification.

In this specification, the major version of the export file format has the value 2 and the minor version has the value 3. Only Oracle may define the meaning and values of new export file format versions.

\textbf{constant\_pool\_count}

The \texttt{constant\_pool\_count} item is a non-zero, positive value that indicates the number of constants in the constant pool.

\textbf{constant\_pool[]}

The \texttt{constant\_pool} is a table of variable-length structures representing various string constants, class names, field names and other constants referred to within the \texttt{ExportFile} structure.

Each of the \texttt{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 \texttt{constant\_pool} table.

\textbf{this\_package}

The value of \texttt{this\_package} must be a valid index into the constant pool table. The \texttt{constant\_pool} entry at that index must be a \texttt{CONSTANT\_Package\_info}\texttt{(5.6.1 \texttt{CONSTANT\_Package})} structure representing the package defined by this ExportFile.

\textbf{referenced\_package\_count}

The \texttt{referenced\_package\_count} item is a value that indicates the number of entries in the \texttt{referenced\_packages} array.

\textbf{referenced\_packages[]}

The \texttt{referenced\_packages} array is a table representing all packages referenced in this export file. Each entry in the \texttt{referenced\_packages} array is an index into the \texttt{constant\_pool} table. The corresponding entry in the \texttt{constant\_pool} array must be \texttt{CONSTANT\_Package}\texttt{(5.6.1 \texttt{CONSTANT\_Package})} structure representing a referenced package. A package is a referenced package if:

- any of its classes are subclassed in this export file,
- any of its interfaces are implemented or extended in this export file,
- any of its classes or interfaces are used in field or method descriptors in this export file.
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 (5.7 Classes and Interfaces) 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` (5.6.1 CONSTANT_Package) 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 public shareable interface declared in this package.¹

5.6 Constant Pool

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

```
cp_info {
    ul tag
    ul 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.

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT_Package</td>
<td>13</td>
</tr>
<tr>
<td>CONSTANT_Classref</td>
<td>7</td>
</tr>
<tr>
<td>CONSTANT_Integer</td>
<td>3</td>
</tr>
<tr>
<td>CONSTANT_Utf8</td>
<td>1</td>
</tr>
</tbody>
</table>

5.6.1 CONSTANT_Package

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

```
CONSTANT_Package_info {
    ul tag
    ul flags
    u2 name_index
    ul minor_version
    ul major_version
}
```

¹ This restriction of exporting only shareable interfaces in non-library packages is imposed by the firewall defined in the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.*
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.

<table>
<thead>
<tr>
<th>Flags</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_LIBRARY</td>
<td>0x01</td>
</tr>
</tbody>
</table>

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. 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. 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 (5.6.1 CONSTANT_Package) 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

---

2 This restriction is imposed by the firewall defined in the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.
binary compatibility between packages. See 4.5 CAP and Package Versions for a description of assigning 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 AID-based Naming).

5.6.2 CONSTANT_Classref
The CONSTANT_Classref_info structure is used to represent a class or interface:

CONSTANT_Classref_info {
    u1 tag
    u2 name_index
}

The items of the CONSTANT_Classref_info structure are the following:

tag

The tag item has the value of CONSTANT_Classref(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 (5.6.4 CONSTANT_Utf8) structure representing a valid fully qualified Java class or interface name. This name is fully qualified since it may represent a class or interface 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.

5.6.3 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. The value of a boolean type is 1 to represent true and 0 to represent false.

5.6.4 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:

```java
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) 0x00−(byte) 0xFF.

5.7 Classes and Interfaces

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

```java
class_info {
  u1 token
  u2 access_flags
  u2 name_index
  u2 export_supers_count
  u2 supers[export_supers_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 Classes and Interfaces) 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 following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside its package</td>
<td>Class, interface</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no subclasses allowed.</td>
<td>Class</td>
</tr>
<tr>
<td>ACC_INTERFACE</td>
<td>0x0200</td>
<td>Is an interface</td>
<td>Interface</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Is abstract; may not be instantiated</td>
<td>Class, interface</td>
</tr>
<tr>
<td>ACC_SHAREABLE</td>
<td>0x0800</td>
<td>Is shareable; may be shared between Java Card applets</td>
<td>Class, interface</td>
</tr>
<tr>
<td>ACC_REMOTE</td>
<td>0x1000</td>
<td>Is remote; may be accessed by Java Card RMI</td>
<td>Class, interface</td>
</tr>
</tbody>
</table>

The `ACC_SHAREABLE` flag indicates whether this class or interface is shareable.³ A class is shareable if it implements (directly or indirectly) the `javacard.framework.Shareable` interface. An interface is shareable if it is or extends (directly or indirectly) the `javacard.framework.Shareable` interface.

³ The `ACC_SHAREABLE` flag is defined to enable Java Card virtual machines to implement the firewall restrictions defined by the `Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition`. 
The **ACC_REMOTE** flag indicates whether this class or interface is remote. The value of this flag must be one if and only if the class or interface satisfies the requirements defined in 2.2.6.1 Remote Classes and Remote Interfaces.

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*.

Since all classes and interfaces represented in an export file are public, the **ACC_PUBLIC** flag must always be set.

All other flag values are reserved. 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_Classref_info** (5.6.2 **CONSTANT_Classref**) structure representing a valid, fully qualified Java class or interface name.

**export_supers_count**

The value of the **export_supers_count** item indicates the number of entries in the **supers** array.

**supers[]**

The **supers** array contains an entry for each public superclass of this class or interface. It does not include package visible superclasses.

For a class, each value in the **supers** array must be a valid index into the **constant_pool** table. The **constant_pool** entry at each value must be a **CONSTANT_Classref_info** structure (5.6.2 **CONSTANT_Classref**) representing a valid, fully-qualified Java class name. Entries in the **supers** array can occur in any order.

For an interface, the **supers** array contains a single value representing a valid index into the **constant_pool** table. The **constant_pool** entry must be a **CONSTANT_Classref_info** structure (5.6.2 **CONSTANT_Classref**) representing the fully-qualified name of the **java.lang.Object** class.

**export_interfaces_count**

The value of the **export_interfaces_count** item indicates the number of entries in the **interfaces** array.

**interfaces[]**

If this **class_info** structure describes a class, the **interfaces** array contains an entry for each public interface implemented by this class. It does not include package-visible interfaces. It does include all public superinterfaces in the hierarchies of public interfaces implemented by this class.
If this `class_info` structure describes an interface, the `interfaces` array contains an entry for each public interface extended by this interface. It does not include package visible interfaces. It does include all public superinterfaces in the hierarchies of public interfaces extended by this interface.

Each value in the `interfaces` array must be a valid index into the `constant_pool` table. The `constant_pool` entry at each value must be a `CONSTANT_Classref_info` structure (5.6.2 `CONSTANT_Classref`) representing a valid, fully-qualified Java interface name. Entries in the interfaces array can occur in any order.

```
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` (5.8 Fields) 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` (5.9 Methods) 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.

```
CAP22_inheritable_public_method_token_count
```

The `CAP22_inheritable_public_method_token_count` item represents the number of public or protected virtual methods inheritable by a subclass defined in a CAP file of format version 2.2 or earlier.

### 5.8 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 0xFF. 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.4

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

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

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Meaning</th>
<th>Used By</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Is public; may be accessed from outside its package.</td>
<td>Any field</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Is protected; may be accessed within subclasses.</td>
<td>Class field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Instance field</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Is static.</td>
<td>Class field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Interface field</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Is final; no further overriding or assignment after initialization.</td>
<td>Any field</td>
</tr>
</tbody>
</table>

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

---

4 Although Java compilers ordinarily replace references to final static fields of primitive types with primitive constants, this functionality is not required.
Since all fields represented in an export file are either public or protected, exactly one of the ACC_PUBLIC or ACC_PROTECTED flag must be set.

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 (5.6.4 CONSTANT_Utf8) 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 (5.6.4 CONSTANT_Utf8) 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).

If this field is a reference-type, the class referenced must be a public class.

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 (5.10.1 ConstantValue Attribute). 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 (5.10.1 ConstantValue Attribute). This must be defined for static final fields of primitive types (boolean, byte, short, and int).

5.9 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 Static Methods and Constructors). If this method is a virtual method, the `token` item represents a virtual method token (4.3.7.6 Virtual Methods). If this method is an interface method, the `token` item represents an interface method token (4.3.7.7 Interface Methods).

**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.

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

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

Since all methods represented in an export file are either public or protected, exactly one of the ACC_PUBLIC or ACC_PROTECTED flag must be set.

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 (5.6.4 CONSTANT_Utf8) 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.
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` (5.6.4 CONSTANT_Utf8) 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 The Export File and Conversion).

All classes referenced in a descriptor must be public classes.

## 5.10 Attributes

Attributes are used in the `field_info` (5.8 Fields) structure of the export file format. All attributes have the following general format:

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

### 5.10.1 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:

```plaintext
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` (5.6.4 CONSTANT_Utf8) 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` (5.6.3 `CONSTANT_Integer`).
6 The CAP File Format

This chapter describes the Java Card converted applet (CAP) file format. A CAP file represents a Java Card application or a library comprising one or more Java packages. A Java Card CAP file may contain only applet packages, only library packages or a combination of applet and library packages. Additionally, both applet and library packages in a Java Card CAP file can be public or private. Each CAP file contains all of the classes and interfaces defined in a Java Card application or library. Java Card Converters must be capable of producing CAP files that conform to the specification provided in this chapter.

6.1 CAP File Overview

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 \texttt{u1}, \texttt{u2} and \texttt{u4} represent an unsigned one-, two and four-byte quantities, respectively. Some \texttt{u1} types are represented as bitfield structures, consisting of arrays of bits. The zeroth 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 \textit{items}. Unlike the fields of a C structure, successive items are stored in the Java Card platform 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 described 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.

A CAP file may be in Compact or Extended Format where the Compact Format CAP file must not contain information for more than one application or library package. A CAP file in Compact format may only
contain 64KB of bytecode information. A CAP file in Extended format may contain 128 blocks containing bytecode information where each block may be of 64KB. See 6.10 Method Component.

Java Card virtual machine implementations must support Compact Format CAP files. Support for Extended Format CAP files is optional.

6.2 Component Model
A Java Card CAP file consists of a set of components. Each component describes a set of elements in the application or library defined, or an aspect of the CAP file. A complete CAP file must contain all of the required components specified in this chapter. Four components are optional: the Applet Component (6.6 Applet Component), Export Component (6.13 Export Component), Debug Component (6.15 Debug Component) and Static Resources Component (6.16 Static Resource Component). The Applet Component is included only if one or more applets are defined in one or more of the packages in the CAP file. The Export Component is included only if classes in other packages, which are not part of the CAP file, may import elements in any of the packages defined. The Debug Component contains all of the data necessary for debugging packages in the CAP file. The Static Resources Component contains all the static resources for the CAP file and must be included if a Java Card application or library contains static resources. A Java Card CAP file may not have a Static Resources Component if the application or library it represents does not have any static resources.

The content of each component defined in a CAP file must conform to the corresponding format specified in this chapter. Components have one of the following general formats (Compact and Extended) as specified in the Header Component (6.4 Header Component). A CAP file in Compact format must only contain components in Compact format except Debug component and Static Resources Component. The Header Component is always in Compact format.

```java
component_compact {  
   u1 tag  
   u2 size  
   u1 info[]  
}
```

```java
component_extended { (since CAP format 2.3)  
   u1 tag  
   u4 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.
Table 6-1: CAP File Component Tags

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPONENT_Header</td>
<td>1</td>
</tr>
<tr>
<td>COMPONENT_Directory</td>
<td>2</td>
</tr>
<tr>
<td>COMPONENT_Applet</td>
<td>3</td>
</tr>
<tr>
<td>COMPONENT_Import</td>
<td>4</td>
</tr>
<tr>
<td>COMPONENT_ConstantPool</td>
<td>5</td>
</tr>
<tr>
<td>COMPONENT_Class</td>
<td>6</td>
</tr>
<tr>
<td>COMPONENT_Method</td>
<td>7</td>
</tr>
<tr>
<td>COMPONENT_StaticField</td>
<td>8</td>
</tr>
<tr>
<td>COMPONENT_ReferenceLocation</td>
<td>9</td>
</tr>
<tr>
<td>COMPONENT_Export</td>
<td>10</td>
</tr>
<tr>
<td>COMPONENT_Descriptor</td>
<td>11</td>
</tr>
<tr>
<td>COMPONENT_Debug (since CAP format 2.2)</td>
<td>12</td>
</tr>
<tr>
<td>COMPONENT_Static_Resources (since CAP format 2.3)</td>
<td>13</td>
</tr>
</tbody>
</table>

Oracle may define additional components in future versions of this Java Card virtual machine specification. It is guaranteed that additional components will have tag values between 14 and 127, inclusive.

6.2.1 Containment in a JAR File
Each CAP file component is represented as a single file. The component file names are enumerated in Table 6-2. These names are not case sensitive. Note that components with extended length have extension “.capx”. Note that Method Component, RefLocation component and Descriptor component with extended length may only be present in the CAP file in Extended format. In Compact format all components, except Debug component, Static Resources component and custom components, must have compact length and must have extension “.cap”.

Table 6-2: CAP File Component File Names

<table>
<thead>
<tr>
<th>Component Type</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPONENT_Header</td>
<td>Header.cap</td>
</tr>
<tr>
<td>COMPONENT_Directory</td>
<td>Directory.cap</td>
</tr>
<tr>
<td>COMPONENT_Applet</td>
<td>Applet.cap</td>
</tr>
<tr>
<td>COMPONENT_Import</td>
<td>Import.cap</td>
</tr>
<tr>
<td>COMPONENT_ConstantPool</td>
<td>ConstantPool.cap</td>
</tr>
<tr>
<td>COMPONENT_Class</td>
<td>Class.cap</td>
</tr>
<tr>
<td>COMPONENT_Method</td>
<td>Method.cap[x]</td>
</tr>
<tr>
<td>COMPONENT_StaticField</td>
<td>StaticField.cap</td>
</tr>
<tr>
<td>COMPONENT_ReferenceLocation</td>
<td>RefLocation.cap[x]</td>
</tr>
<tr>
<td>COMPONENT_Export</td>
<td>Export.cap</td>
</tr>
<tr>
<td>COMPONENT_Descriptor</td>
<td>Descriptor.cap[x]</td>
</tr>
<tr>
<td>COMPONENT_Debug (since CAP format 2.2)</td>
<td>Debug.cap[x]</td>
</tr>
<tr>
<td>COMPONENT_Static_Resources (since CAP format 2.3)</td>
<td>StaticResources.capx</td>
</tr>
</tbody>
</table>
All CAP file components are stored in a JAR file. As described in 4.1.3 JAR File Container, the path to the CAP file component files in a JAR file consists of a directory called javacard.

Name of the Debug Component in Compact format must have extension “.cap” if the size of info item is less than 65535 bytes. Size item of Debug Component in Compact format shall always be of type u2. The Debug Component in Compact format may also contain 65535 or more bytes in which case the size item shall have value of 0xFFFF and the name of the Debug component in the jar file must have extension “.capx”. Name of the Debug component in Extended format must always have extension “.capx”.

For JAR files in Compact format, the directory “javacard” 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.

For JAR files in Extended Format, the directory “javacard” is in a subdirectory representing the application or library’s directory. For example, the CAP file component files of the application HelloWorld are located in the subdirectory com/oracle/HelloWorld/javacard.

Other files, including other CAP files, may also reside in a JAR file that contains CAP file component files.

The JAR file format provides a vehicle suitable for the distribution of CAP file components. It is not intended or required that the JAR file format be used as the load file format for loading CAP file components onto a Java Card technology-enabled device. See 6.3 Installation for more information.

The name of a JAR file containing CAP file components is not defined as part of this specification. The naming convention used by the Oracle Java Card Converter Tool is to append .cap to the application or library name. For example, the CAP file produced for the library JavaLoyalty would be named JavaLoyalty.cap.

6.2.2 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 AID-based Naming) 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 (6.5 Directory Component).

The new component must conform to one of the general component formats defined in this chapter, either compact or extended as defined in the Header Component, 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 JAR File Container. That is, the file containing the new component must be located in the `<application_or_library_directory>/javacard` or `<package_directory>/javacard` subdirectory of the JAR file and must have the extension `.capx` or `.cap` based on Extended or Compact format of the CAP file respectively.

### 6.3 Installation

Installing CAP file components onto a Java Card technology-enabled device entails communication between a Java Card technology-enabled terminal and that device. While it is beyond the scope of this specification to define a load file format or installation protocol between a terminal and a device, the CAP file component order shown below is a reference load order suitable for an implementation with a simple memory management model on a limited memory device.\(^5\)

**Reference Component Install Order:**

- COMPONENT_Header
- COMPONENT_Directory
- COMPONENT_Import
- COMPONENT_Applet
- COMPONENT_Class
- COMPONENT_Method
- COMPONENT_StaticField
- COMPONENT_Export
- COMPONENT_ConstantPool
- COMPONENT_ReferenceLocation
- COMPONENT_Static_Resources
- COMPONENT_Descriptor (optional)

The component type COMPONENT_Debug is not intended for download to the device. It is intended to be used off-card in conjunction with a suitably instrumented Java Card virtual machine.

### 6.4 Header Component

The Header Component contains general information about this CAP file and the public packages it defines. It is described by one of the following variable-length structures:

```java
header_component_compact {
    ul tag
    u2 size
    u4 magic
    ul CAP_Format_minor_version
```

\(^5\) Both the Java Card Forum and Global Platform specification have adopted this component load order as a standard to enhance interoperability. In both cases, loading the Descriptor Component is optional. Furthermore, the Global Platform specification defines the format of packets (APDUs) used during installation.
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`.

`CAP_Format_minor_version`, `CAP_Format_major_version`

The `CAP_Format_minor_version` and `CAP_Format_major_version` items are the minor and major version numbers of the format used for this CAP file. If a CAP file has the major version number of $M$ and minor version number of $m$, the version of the CAP file’s format is $M.m$.

A change in the major version number indicates a major incompatibility change, one that requires a fundamentally different Java Card virtual machine. A Java Card virtual machine is not required to support CAP files with different major version numbers. A Java Card virtual machine is required to
support CAP files having a given major version number and all valid minor version numbers in the range 0 through some particular minor_version where a valid minor version number is a minor version number that has been defined in a version of the Java Card virtual machine specification.

In this specification, the major version of the CAP file format has the value 2 and the minor version has the value 3. A Java Card Virtual Machine must support minor version values between 1 and 3. Only Oracle Corp. may define the meaning and values of new CAP file format versions.

flags

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

Table 6-3: CAP File Flags

<table>
<thead>
<tr>
<th>Flags</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_INT</td>
<td>0x01</td>
</tr>
<tr>
<td>ACC_EXPORT</td>
<td>0x02</td>
</tr>
<tr>
<td>ACC_APPLET</td>
<td>0x04</td>
</tr>
<tr>
<td>ACC_EXTENDED</td>
<td>0x08</td>
</tr>
</tbody>
</table>

The ACC_INT flag has the value of one if the Java int type is used by at least one of the packages in this CAP file. The int type is used if one or more of the following is present:

- A parameter to a method of type int
- A parameter to a method of type int array
- A local variable of type int
- A local variable of type int array
- A field of type int
- A field of type int array
- An instruction of type int, or
- An instruction of type int array

Otherwise the ACC_INT flag has the value of 0.

The ACC_EXPORT flag has the value of one if an Export Component (6.13 Export Component) 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 (6.6 Applet Component) is included in this CAP file. Otherwise it has the value of 0.

All other bits in the flags item not defined in Table 6-3 are reserved for future use. Their values must be zero.
The ACC_EXTENDED flag has the value of one if the CAP file is in Extended format. In this case the Method Component (6.10 Method Component), Reference Location component (6.12 Reference Location Component), Descriptor Component (6.14 Descriptor Component), Debug Component (6.15 Debug Component) and all custom components in the CAP file must be in the Extended Format. Otherwise it has the value of 0.

package

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

```plaintext
class 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 CAP and Package Versions for a description of assigning 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 platform name of the package. See ISO 7816-5 for the definition of an AID (also see 4.2 AID-based Naming).

- **package_name**

  The package_name item in header_component_compact, describes the name of the package defined in this CAP file. It is represented as a package_name_info[] structure:

```plaintext
class package_name_info {  
    (since CAP format 2.2)  
    u1 name_length  
    u1 name[name_length]  
}
```
The items in the `package_name_info[]` structure are as follows:

`name_length`

The `name_length` item is the number of bytes used in the name item to represent the name of this package in UTF-8 format. The value of this item may be zero if and only if the package does not define any remote interfaces or remote classes.

`name[]`

The `name[]` item is a variable length representation of the fully qualified name of this package in UTF-8 format. The fully qualified name is represented in internal form as described in the *Java Virtual Machine Specification*.

`CAP_minor_version, CAP_major_version`

The `CAP_minor_version` and `CAP_major_version` items in `header_component_extended` are the minor and major version numbers of this CAP file. These values uniquely identify this CAP file. See 4.5 CAP and Package Versions for a description of assigning and using version numbers.

`CAP_AID_length`

The `CAP_AID_length` item in `header_component_extended`, represents the number of bytes in the `CAP_AID` item. Valid values are between 5 and 16, inclusive.

`CAP_AID[]`

The `CAP_AID` item in `header_component_extended`, represents the Java Card platform name of the CAP file. See ISO 7816-5 for the definition of an AID (also see 4.2 AID-based Naming).

`package_count`

The `package_count` item in the `header_component_extended` structure represents the number of entries in the `packages` table. The value of the `package_count` item must be greater than 0.

`packages`

The `packages` item in `header_component_extended`, is an array representing all public packages in this CAP file. Each entry in the array is represented as a `package_info` structure.

`package_names`

The `package_names` item in `header_component_extended`, describes the names of all the packages defined in this CAP file. Each name is represented as a `package_name_info[]` structure.

**6.5 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 (6.6 Applet Component), Export
Component (6.13 Export Component), Debug Component (6.15 Debug Component), or for Extended format, the Static Resource Component (Static Resource Component) 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 for CAP file in Compact and Extended format is described by the following variable-length structures:

```
directory_component_compact {  
  ul tag  
  u2 size  
  component_size_info_compact component_sizes (since CAP format 2.3)  
  static_field_size_info static_field_size  
  ul import_count  
  ul applet_count  
  ul custom_count  
  custom_component_info_compact custom_components[custom_count]  
}

directory_component_extended { (since CAP format 2.3)  
  ul tag  
  u2 size  
  component_size_info_extended component_sizes  
  static_field_size_info static_field_size  
  ul import_count  
  ul applet_count  
  ul method_component_block_count  
  ul custom_count  
  custom_component_info_extended custom_components[custom_count]  
}
```

The items in these structures 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 a structure of type `component_size_info_compact` or `component_size_info_extended` based on the Compact or Extended format of the CAP file. It represents the number of bytes in each of the components in this CAP file.
component_size_info_compact {
   u2 Header_Component_Size
   u2 Directory_Component_Size
   u2 Applet_Component_Size
   u2 Import_Component_Size
   u2 Constant_Pool_Component_Size
   u2 Class_Component_Size
   u2 Method_Component_Size
   u2 Static_Field_Component_Size
   u2 Reference_Location_Component_Size
   u2 Export_Component_Size
   u2 Descriptor_Component_Size
   u2 Debug_Component_Size (since CAP format 2.2)
   u4 Static_Resource_Component_Size (since CAP format 2.3)
}

component_size_info_extended { (since CAP format 2.3)
   u2 Header_Component_Size
   u2 Directory_Component_Size
   u2 Applet_Component_Size
   u2 Import_Component_Size
   u2 Constant_Pool_Component_Size
   u2 Class_Component_Size
   u4 Method_Component_Size
   u2 Static_Field_Component_Size
   u4 Reference_Location_Component_Size
   u2 Export_Component_Size
   u4 Descriptor_Component_Size
   u4 Debug_Component_Size
   u4 Static_Resource_Component_Size
}

The value of an entry in the component_size_info_compact or component_size_info_extended structures is zero for components not included in this CAP file. Components that may not be included are the Applet Component (6.6 Applet Component), the Export Component (6.13 Export Component), the Debug Component (6.15 Debug Component), and the Static Resource Component (6.15 Static Resource Component). For all other components the value is greater than zero.

Value for Debug_Component_Size in component_size_info_compact must be 0xFFFF if size of the info item of the Debug Component (6.15 Debug Component) is greater than or equal to 65535.

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 (6.11 Static Field Component). It represents the total number of bytes in the static fields defined in all packages in this CAP file, 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 (6.11 Static Field Component). It represents the number of arrays initialized in all of the <clinit> methods in all the packages in this CAP file.

**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 (6.11 Static Field Component). It is the total number of bytes in all of the arrays initialized in all of the <clinit> methods in all the packages in this CAP file.

**import_count**

The import_count item indicates the number of packages imported by classes and interfaces in this CAP file. This item has the same value as the count item in the Import Component (6.7 Import Component).

**applet_count**

The applet_count item indicates the number of applets defined in packages in this CAP file. If an Applet Component (6.6 Applet Component) 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 (6.6 Applet Component).

**method_component_block_count**

The method_component_block_count item indicates the number of method component blocks in the Extended Method Component (6.10 Method Component). The value of the
method_component_block_count item is the same as the value of the
method_component_block_count item in the Method Component (6.10 Method Component).

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_compact or custom_component_info_extended structures based on Compact or Extended length format of the CAP file. 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_compact and custom_component_info_extended structures are defined as:

custom_component_info_compact {
  u1 component_tag
  u2 size
  u1 AID_length
  u1 AID[AID_length]
}

custom_component_info_extended { (since CAP format 2.3)
  u1 component_tag
  u4 size
  u1 AID_length
  u1 AID[AID_length]
}

The items in entries of these structures 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 platform name of the component. See ISO 7816-5 for the definition of an AID (4.1 Java Card Platform File Formats).

Each component is assigned an AID conforming to the ISO 7816-5 standard. Beyond that, there are no constraints on the value of an AID of a custom component.

6.6 Applet Component

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

The Applet Component is described by one of the following variable-length structures based on Compact or Extended format of CAP file:

```
applet_component_compact {
    u1 tag
    u2 size
    u1 count
      {u1 AID_length
       u1 AID[AID_length]
       u2 install_method_offset
       } aplets[count]
}
```

```
applet_component_extended { (since CAP format 2.3)
    u1 tag
    u2 size
    u1 count
      {u1 AID_length
       u1 AID[AID_length]
       u1 install_method_component_block_index
       u2 install_method_offset
       } aplets[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.

count

The count item indicates the number of applets defined by all the packages in this CAP file. The value of the count item must be greater than zero.

applets[]

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

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 platform name of the applet.

Each applet is assigned an AID conforming to the ISO 7816-5 standard (4.2 AID-based Naming). 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_component_block_index

The install_method_component_block_index item in applet_component_extended structure represents the index into the blocks array of Method Component (6.10 Method Component). The method_component_block at that index contains the install method for this applet.

install_method_offset

The value of the install_method_offset item in applet_component_compact structure, must be a 16-bit offset into the info item of the Method Component (6.10 Method Component).

The value of the install_method_offset item in applet_component_extended structure must be a 16-bit offset into the method_component_block in the blocks array of Method Component
(6.10 Method Component) at index represented by
install_method_component_block_index.

The item at that offset must be a method_info structure that represents the static install(byte[], short, byte) method of the applet.\(^7\) 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.

### 6.7 Import Component

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

```plaintext
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 128, inclusive.

packages[]

The packages item represents a table of variable-length package_info structures as defined for package under 6.4 Header Component. The table contains an entry for each of the packages referenced in the CAP file, not including the packages defined in this CAP file.

---

\(^7\) Restrictions placed on the install(byte[], short, byte) method of an applet are imposed by the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.
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 CAP and Package Versions for a description of assigning and using package version numbers.

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

### 6.8 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 (6.10 Method Component) 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 (6.9 Class Component), Method Component (6.10 Method Component), and Static Field Component (6.11 Static Field Component). The Import Component (6.7 Import Component) is also accessed using a package token (4.3.7.1 Package) 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:

```java
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:
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.

### Table 6-4: CAP File Constant Pool Tags

<table>
<thead>
<tr>
<th>Constant Type</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT_Classref</td>
<td>1</td>
</tr>
<tr>
<td>CONSTANT_InstanceOfFieldref</td>
<td>2</td>
</tr>
<tr>
<td>CONSTANT_VirtualMethodref</td>
<td>3</td>
</tr>
<tr>
<td>CONSTANT_SuperMethodref</td>
<td>4</td>
</tr>
<tr>
<td>CONSTANT_StaticFieldref</td>
<td>5</td>
</tr>
<tr>
<td>CONSTANT_StaticMethodref</td>
<td>6</td>
</tr>
</tbody>
</table>

Java Card platform constant types (“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 platform constant pool (“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. 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_InstanceOfFieldref (6.8.2 CONSTANT_InstanceOfFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref) 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.

The first entry in the constant pool cannot be an exception handler class that is referenced by a catch_type_index of an exception_handler_info structure. In such a case the value of the

---

8 The constant pool index parameter of an invokespecial bytecode is to a CONSTANT_StaticMethodref when the method referenced is a constructor or a private instance 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.
catch_type_index would be equal to 0, but the value of 0 in a catch_type_index is reserved to indicate an exception_handler_info structure that describes a finally block.

6.8.1 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 CAP file, the structure represents an internal_class_ref and the high bit of the structure is zero. If the class or interface is defined in imported 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 (6.9 Class Component) 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 be 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 Package) defined in the Import Component (6.7 Import Component) 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 Classes and Interfaces) of the referenced class or interface. It has the value of the class token of the class as defined in the Exportfile 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.

6.8.2 CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref

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

CONSTANT_InstanceFieldref_info {
   ul tag
   class_ref class
   ul token
}

CONSTANT_VirtualMethodref_info {
   ul tag
   class_ref class
   ul token
}

CONSTANT_SuperMethodref_info {
   ul tag
   class_ref class
   ul 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 (6.8.1 CONSTANT_Classref). If the referenced
class is defined in this CAP file, 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 be the class that
defines the method that contains the Java language-level super invocation.

token

The token item in the CONSTANT_InstanceFieldref_info structure represents an instance
field token (4.3.7.5 Instance Fields) 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 Virtual Methods) 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 Virtual Methods) 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 superclass of the class that contains a
definition of the virtual method must also be defined in this package.

6.8.3 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 method_info_block_index (since CAP format 2.3)
         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

The **static_field_ref item represent a reference to a static field.**
If the referenced item is defined in this CAP file the structure represents an `internal_ref` and the high bit of the structure is zero. If the referenced item is defined in another CAP file 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 defined in this CAP file. The items in the structure are:

- **padding**

The `padding item` of a `CONSTANT_StaticFieldref_info structure` is equal to 0.

- **method_info_block_index**

The `method_info_block_index item` of a `CONSTANT_StaticMethodref_info structure` represents the index in the blocks array of Method Component (6.10 Method Component) containing the referenced method.

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

- **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 (6.11 Static Field Component) to this static field.

The `offset item` of a `CONSTANT_StaticMethodref_info structure` represents a 16-bit offset into the `method_component_block` in the blocks array of Method Component (6.10 Method Component) at index represented by `method_info_block_index`. The `method_info structure` at `offset` 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 Package) defined in the Import Component (6.7 Import Component) 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 Classes and Interfaces) 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 Static Fields) 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 Static Methods and Constructors) as defined in the Export file of the imported package. It has the value of the token of the referenced method.

6.9 Class Component

The Class Component describes each of the classes and interfaces defined in this CAP file. 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 CAP file are included in the Descriptor Component (6.14 Descriptor Component).

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 (6.10 Method Component) 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 (6.8 Constant Pool Component), Export Component (6.13 Export Component), Descriptor Component (6.14 Descriptor Component) and Debug Component (6.15 Debug Component) 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 structures based on Compact or Extended format of the CAP file:

```java
class_component_compact {
    u1 tag
    u2 size
    u2 signature_pool_length   (since CAP format 2.2)
    type_descriptor signature_pool[](since CAP format 2.2)
    interface_info interfaces[]
    class_info_compact classes[]
}

class_component_extended { (since CAP format 2.3)
    u1 tag
    u2 size
    u2 signature_pool_length
    type_descriptor signature_pool[]
    interface_info interfaces[]
    class_info_extended classes[]
}
```

The items in the `class_component_compact` and `class_component_extended` structures 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.

**signature_pool_length**

The **signature_pool_length** item indicates the number of bytes in the `signature_pool[]` item. The value of the `signature_pool_length` item must be zero if none of the packages in this CAP file define any remote interfaces or remote classes.

**signature_pool[]**
The **signature_pool[]** item represents a list of variable-length **type_descriptor** structures. These descriptors represent the signatures of the remote methods.

```java
interfaces[]
```

The **interfaces[]** item represents an array of **interface_info** structures. Each interface defined in this CAP file 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.

```java
classes[]
```

The **classes[]** item represents a table of variable-length **class_info_compact** or **class_info_extended** structures based on Compact or Extended format of CAP file. Each class defined in this CAP file 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.

### 6.9.1 type_descriptor

The **type_descriptor** structure represents the type of a field or the signature of a method.

```java
type_descriptor { (since CAP format 2.2)
    ul nibble_count;
    ul type[(nibble_count+1) / 2];
}
```

The **type_descriptor** structure contains the following elements:

- **nibble_count**

  The **nibble_count** value represents the number of nibbles required to describe the type encoded in the **type[]** array.

- **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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Void</td>
<td>0x1</td>
</tr>
<tr>
<td>Boolean</td>
<td>0x2</td>
</tr>
<tr>
<td>Byte</td>
<td>0x3</td>
</tr>
<tr>
<td>Short</td>
<td>0x4</td>
</tr>
<tr>
<td>Int</td>
<td>0x5</td>
</tr>
<tr>
<td>Reference</td>
<td>0x6</td>
</tr>
<tr>
<td>arrayof boolean</td>
<td>0xA</td>
</tr>
</tbody>
</table>
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 (6.8.1 CONSTANT_Classref). For example, a field of type reference to p1.c1 in a CAP file defining package p0 is described as:

```
<table>
<thead>
<tr>
<th>Nibble</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x6</td>
<td>Reference</td>
</tr>
<tr>
<td>1</td>
<td>&lt;p1&gt;</td>
<td>package token (high bit on)</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>&lt;c1&gt;</td>
<td>class token</td>
</tr>
<tr>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>0x0</td>
<td>Padding</td>
</tr>
</tbody>
</table>
```

The following are examples of the array types:

```
<table>
<thead>
<tr>
<th>Nibble</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xB</td>
<td>array of byte</td>
</tr>
<tr>
<td>1</td>
<td>0x0</td>
<td>Padding</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Nibble</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xE</td>
<td>array of reference</td>
</tr>
<tr>
<td>1</td>
<td>&lt;p1&gt;</td>
<td>package token (high bit on)</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>&lt;c1&gt;</td>
<td>class token</td>
</tr>
<tr>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>0x0</td>
<td>Padding</td>
</tr>
</tbody>
</table>
```

Method signatures are encoded in the same way, with the return type of the method encoded at the end of the sequence of nibbles. The return type is encoded in as many nibbles as required to represent it. For example:
Table 6-9: Encoded Method Signature (J/V)

<table>
<thead>
<tr>
<th>Nibble</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x1</td>
<td>Void</td>
</tr>
<tr>
<td>1</td>
<td>0x0</td>
<td>Padding</td>
</tr>
</tbody>
</table>

Table 6-10: Encoded Method Signature (Lp1.ci;)S

<table>
<thead>
<tr>
<th>Nibble</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x6</td>
<td>Reference</td>
</tr>
<tr>
<td>1</td>
<td>&lt;p1&gt;</td>
<td>package token (high bit on)</td>
</tr>
<tr>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>&lt;c1&gt;</td>
<td>class token</td>
</tr>
<tr>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>0x4</td>
<td>Short</td>
</tr>
</tbody>
</table>

6.9.2 interface_info, class_info_compact and class_info_extended

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:

Note: Below, for interface_name_info interface_name the interface_name[] item is required if the value of ACC_REMOTE is one. This item must be omitted otherwise. See the description of this field for more information. For remote_interface_info remote_interfaces the remote_interfaces item is required if the value of ACC_REMOTE is one. This item must be omitted otherwise. See the description of this field for more information.

interface_info {
    ul bitfield {
        bit[4] flags
        bit[4] interface_count
    }
    class_ref superinterfaces[interface_count]
    interface_name_info interface_name
}

class_info_compact {
    ul bitfield {
        bit[4] flags
        bit[4] interface_count
    }
    class_ref super_class_ref
    ul declared_instance_size
    ul first_reference_token
    ul 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]
remote_interface_info remote_interfaces (since CAP format 2.2)
u1 public_virtual_method_token_mapping[public_method_table_count]
  (since CAP format 2.3)
u1 CAP22_inheritable_public_method_token_count
  (since CAP format 2.3)
}

class_info_extended { (since CAP format 2.3)
u1 bitfield {
    bit[4] flags
    bit[4] interface_count
}
class_ref super_class_ref
u1 declared_instance_size
u1 first_reference_token
u1 reference_count
u1 public_method_table_base
u1 public_method_table_count
u1 package_method_table_base
u1 package_method_table_count
method_block_info
    public_virtual_method_table[public_method_table_count]
method_block_info
    package_virtual_method_table[package_method_table_count]
implemented_interface_info interfaces[interface_count]
remote_interface_info remote_interfaces
u1 public_virtual_method_token_mapping[public_method_count]
u1 CAP22_inheritable_public_method_token_count
}

6.9.2.1 interface_info, class_info_compact and class_info_extended Shared Items

flags

The flags item is a mask of modifiers used to describe this interface or class. Valid values are shown in the following table:
Table 6-11: CAP File Interface and Class Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_INTERFACE</td>
<td>0x8</td>
</tr>
<tr>
<td>ACC_SHAREABLE</td>
<td>0x4</td>
</tr>
<tr>
<td>ACC_REMOTE</td>
<td>0x2</td>
</tr>
</tbody>
</table>

The **ACC_INTERFACE** flag indicates whether this structure represents an interface or a class. The value must be one if it represents an `interface_info` structure and zero if it represents a `class_info_compact` or `class_info_extended` 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 extends that interface directly or indirectly.

The **ACC_SHAREABLE** flag in a `class_info_compact` or `class_info_extended` structure indicates whether this class is shareable. 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 **ACC_REMOTE** flag indicates whether this class or interface is remote. The value of this flag must be one if and only if the class or interface satisfies the requirements defined in 2.2.6.1 Remote Classes and Remote Interfaces.

All other flag values are reserved. 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 direct and indirect superinterfaces of this interface. Indirect superinterfaces are the set of superinterfaces of the direct superinterfaces. Valid values are between 0 and 14, inclusive.

The `interface_count` item of the `class_info_compact` and `class_info_extended` structures 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.

**6.9.2.2 interface_info Items**

**superinterfaces[]**

---

9 A Java Card virtual machine uses the ACC_SHAREABLE flag to implement the firewall restrictions defined by the *Runtime Environment Specification, Java Card Platform, v3.0.6, Classic Edition.*
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 (6.8.1 CONSTANT_Classref). This array is empty if this interface has no superinterfaces. Both direct and indirect superinterfaces are represented in the array. Class Object is not included.

interface_name[]

The interface_name[] item represents interface name information required if the interface is remote. The interface_name[] item is defined by a table of variable-length interface_name_info structures. If the value of the ACC_REMOTE flag is zero, the structure is defined as:

interface_name_info {
}

If the value of the ACC_REMOTE flag is one, the structure is defined as:

interface_name_info {
    ul interface_name_length
    ul interface_name[interface_name_length]
}

The values in the interface_name_info structure are defined as follows:

interface_name_length

The interface_name_length item is the number of bytes in interface_name[] item.

interface_name

The item is a variable length representation of the name of this interface in UTF-8 format.

6.9.2.3 class_info_compact and class_info_extended Items

super_class_ref

The super_class_ref item of the class_info_compact and class_info_extended structures 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 (6.8.1 CONSTANT_Classref).

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_compact and class_info_extended structures represents the number of 16-bit cells required to represent the instance fields declared by 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_compact and class_info_extended structures represents the instance field token (4.3.7.5 Instance Fields) 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_compact and class_info_extended structures 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_compact and class_info_extended structures is equal to the virtual method token value (4.3.7.6 Virtual Methods) 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_compact and class_info_extended structures 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_compact and class_info_extended structures 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_compact` and `class_info_extended` structures is equal to the virtual method token value (4.3.7.6 Virtual Methods) 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_compact` and `class_info_extended` structures 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_compact` and `class_info_extended` structures 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.

In `class_info_compact` structure, the 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 (6.10 Method Component) to the `method_info` structure representing the method. Entries that represent methods defined or declared in an imported package contain the value 0xFFFF.

In `class_info_extended` structure, the entries in the `public_virtual_method_table[]` array are of type `method_block_info` structure that represent methods defined or declared in the package containing this class. Entries that represent methods defined or declared in an imported package contain the value 0xFFFF for the `method_offset` item of `method_block_info` structure and the value 0xFF for `method_component_block_index` item of `method_block_info` structure.

Entries for methods that are declared abstract 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 the package containing this class. 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.

In `class_info_compact` structure, the entries in `package_virtual_method_table[]` array contain offsets into the info item of the Method Component (6.10 Method Component) to the `method_info` structure representing the method.
In class_info_extended structure, the entries in package_virtual_method_table[] array are of type method_block_info structure that represent methods defined or declared in the package containing this class.

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.

6.9.2.4 method_block_info
This structure is defined as:

```java
method_block_info { (since CAP format 2.3)
    u1 method_component_block_index
    u2 method_offset
}
```

The method_block_info structure is defined as:

- `method_component_block_index`

  The method_component_block_index item represents the index into the blocks array of Method Component (6.10 Method Component). The method_component_block at that index contains the referenced method.

- `method_offset`

  The value of the method_offset item must be a 16-bit offset into the method_component_block in the blocks array of Method Component (6.10 Method Component) at index represented by method_component_block_index. The method_info structure at offset must represent the referenced method.

`interfaces[]`

The interfaces item of the class_info_compact and class_info_extended structures represents a table of variable-length implemented_interface_info structures. The table must contain an entry for each of the directly 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.

Given the declarations below, the number of entries for class c0 is 1 and the entry in the interfaces array is i0. The 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 must not include interface i0, which is implemented only by the superclass of c1.

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

remote_interfaces

The remote_interfaces item represents information required if this class or any of its super classes implements a remote interface. This item must be omitted if the ACC_REMOTE flag has a value of zero. The remote_interfaces item is defined by a remote_interface_info structure.

6.9.2.5 implemented_interface_info
The implemented_interface_info structure is defined as follows:

implemented_interface_info {
    class_ref interface
    ul count
    ul 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 (6.8.1 CONSTANT_Classref). 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 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 Interface Methods) 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, not within the scope of this class.

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

For information on runtime resolution of an interface method see 7.5.54.1 Interface Method Resolution.
6.9.2.6 remote_interface_info

If the value of the ACC_REMOTE flag is zero, this structure is defined as:

remote_interface_info { (since CAP format 2.2) }

If the value of the ACC_REMOTE flag is one, this structure is defined as:

remote_interface_info { (since CAP format 2.2)
  u1 remote_methods_count
  remote_method_info remote_methods[remote_methods_count]
  u1 hash_modifier_length
  u1 hash_modifier[hash_modifier_length]
  u1 class_name_length
  u1 class_name[class_name_length]
  u1 remote_interfaces_count
  class_ref remote_interfaces[remote_interfaces_count]
}

The remote_interface_info structure is defined as:

remote_methods_count

The remote_methods_count item indicates the number of entries in the remote_methods array.

remote_methods[]

The remote_methods item of the class_info structure is an array of remote_method_info structures that maps each remote method available in the class to its hash code and its type definition in the signature_pool[]. The methods are listed in numerically ascending order of hash values.

The remote_method_info structure is defined as follows:

remote_method_info { (since CAP format 2.2)
  u2 remote_method_hash
  u2 signature_offset
  u1 virtual_method_token
}

The items in the remote_method_info structure are defined as follows:

remote_method_hash

The remote_method_hash item contains a two-byte hash value for the method. The hash value is computed from the simple (not fully qualified) name of the method concatenated with its method
descriptor. The representation of the method descriptor is the same as in a Java class file. See the specification described in *The Java Virtual Machine Specification* (§4.3.3).

The hash value uniquely identifies the method within the class.

The hash code is defined as the first two bytes of the SHA-1 message digest function performed on the hash_modifier[] item described below followed by the name of the method followed by the method descriptor representation in UTF-8 format. Rare hash collisions are averted automatically during package conversion by adjusting the anti-collision string.

**signature_offset**

The signature_offset item contains an offset from the signature_pool item of the info item of the Class Component to the variable-length type descriptor structure inside the signature_pool[] item. This structure represents the signature of the remote method.

**virtual_method_token**

The virtual_method_token item is the virtual method token of the remote method in this class.

**hash_modifier_length**

The hash_modifier_length item is the number of bytes in the following hash_modifier item. The value of this item must be zero if an anti-collision string is not required.

**hash_modifier[]**

The hash_modifier[] item is a variable length representation of the anti-collision string in UTF-8 format.

**class_name_length**

The class_name_length item is the number of bytes used in the class_name[] item.

**class_name[]**

The class_name[] item is a variable length representation of the name of this class in UTF-8 format.

**remote_interfaces_count**

The remote_interfaces_count item is the number of interfaces listed in the following remote_interfaces[] item.

**remote_interfaces[]**

The remote_interfaces[] item is a variable length array of class_ref items. It represents the remote interfaces implemented by this class. The remote interfaces listed in this array, together with
their superinterfaces must be the complete set of remote interfaces implemented by this class and all its superclasses.

Each entry has the form of a class_ref structure. Each class_ref structure must reference an interface_info structure representing a remote interface implemented by this class.

The entries in the remote_interfaces[] array must be ordered such that all remote interfaces from the same package are listed consecutively.

6.9.2.7 public_virtual_method_token_mapping

The public_virtual_method_token_mapping[] item of the class_info structure represents an array mapping public and protected virtual method tokens of methods overridden or declared in this class to the corresponding token value in the super class. The value of an index into this table must be equal to the value of the virtual method token of the indicated method in this class. The number of entries in this array is public_method_count where public_method_count equals public_method_table_base plus public_method_table_count.

The entries in the public_virtual_method_token_mapping[] array that represent methods declared by this class must contain the value 0xFF. The entries in the public_virtual_method_token_mapping[] array that represent methods declared by any one of the super classes of this class must contain the value of the token in the direct super class at the time of conversion of this class.

The same virtual method can have different tokens in different classes. Given a class C2 and its direct super-class C1, the token T2 in class C2 denotes the same method as token T1 in class C1 when T1=C2.public_virtual_method_token_mapping[T2] and T1 is not 0xFF. This relation can be extended by reflexivity, symmetry, and transitivity to any classes C1 and C2.

The content of the table public_virtual_method_token_mapping for a class C2 defined in a CAP file of format 2.2 or earlier is implicitly defined as:

- For T < inherited_public_method_count,
  public_virtual_method_token_mapping[T] = T
- For inherited_public_method_count <= T < public_method_count,
  public_virtual_method_token_mapping[T] = 0xFF

where inherited_public_method_count is defined by using the immediate super-class C1 of C2 and depends on the format of the CAP file defining C1:

- If CAP file defining C1 has format 2.2 or earlier,
  inherited_public_method_count = C1.public_method_count
- If CAP file defining C1 has format 2.3,
  inherited_public_method_count = C1.CAP22_inheritable_public_method_token_count
For information on runtime resolution of a virtual method see 7.5.57.1 Virtual Method Resolution.

CAP22_inheritable_public_method_token_count

The CAP22_inheritable_public_method_token_count item represents the number of public or protected virtual methods inheritable by a subclass defined in a CAP file of format version 2.2 or earlier.

6.10 Method Component

The Method Component describes each of the methods declared in this CAP file, excluding <clinit> methods and interface method declarations. Abstract methods defined by classes (not interfaces) are included. 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 optimized for size and therefore 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 (6.14 Descriptor Component). Among other information, the Descriptor Component contains the location and number of bytecodes for each method in the Method Component. This information can be used to parse the methods in the Method Component.

Instructions and exception handler catch types in the Method Component reference entries in the Constant Pool Component (6.8 Constant Pool Component). No other CAP file components, including the Method Component, are referenced by the elements in the Method Component.


The Method Component is represented by one of the following structures based on Compact or Extended format of the CAP file:

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

```
method_component_extended { (since CAP format 2.3)
```


u1 tag
u4 size
u1 method_component_block_count
u4 method_component_block_offsets[method_component_block_count]
method_component_block blocks[method_component_block_count]
)

The items in these structures 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.

method_component_block_count

The method_component_block_count item in method_component_extended structure indicates the number of method_component_block entries in the blocks array. The value of method_component_block_count item must be between 0 and 127, inclusive.

method_component_block_offsets

The method_component_block_offsets[] item of the method_component_extended structure represents an array of offsets into the info item of the method component for each method_component_block. The number of entries in this array must be the same as the value of method_component_block_count.

blocks[]

The blocks[] item of the method_component_extended structure represents an array of method_component_block items. The number of entries in this array must be the same as the value of method_component_block_count.

6.10.1 method_component_block

A method_component_extended may contain between 1 and 127 method_component_block items. Each block can have a maximum size of 65535 bytes. A block must contain all the information needed to execute any method contained in that block. This means that every single method_info structure and all its corresponding exception handlers must be contained in one method_component_block. If adding a method to a method_component_block would exceed the maximum size for the block, the CAP file Converter tool must create a new method_component_block and add this method and all its corresponding exception handlers to that new block.
The method_component_block structure is defined as follows:

```java
method_component_block { (since CAP format 2.3)
    u1 handler_count
    exception_handler_info exception_handlers[handler_count]
    method_info methods[]
}
```

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 CAP file.

Entries in the exception_handlers array are sorted in ascending order by the offset to the handler of the exception handler. Smaller offset values occur first in the array. This ordering constraint ensures that the first match found when searching for an exception handler is the correct match.

There are two consequences of this ordering constraint. First, a handler that is nested with the active range (try block) of another handler occurs first in the array. Second, when multiple handlers are associated with the same active range, they are ordered as they occur in a method. This is consistent with the ordering constraints defined for Java class files. An example is shown in 6.10.2 Exception Handler Example.

6.10.2 Exception Handler Example

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.

```java
try {
    ...
    try {
        ...
        } catch (NullPointerException e) { // first
            ...
        }
    ...
    } catch (Exception e) { // second
        ...
    } finally { // third
        ...
```
try {
...
} catch (SecurityException e) {  // fourth
...
}
methods[]

6.10.3 exception_handler_info

The exception_handler_info structure is defined as follows:

```
exception_handler_info {
    u2 start_offset
    u2 bitfield {
        bit[1] stop_bit
        bit[15] 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 `start_offset` and `active_length` pair indicate the active range (try block) in an exception handler. The `start_offset` item indicates the beginning of the active range while the `active_length` item indicates the number of bytes contained in the active range.

`end_offset` is defined as `start_offset` plus `active_length`.

In method_component_compact structure, the `start_offset` item and `end_offset` are byte offsets into the info item of the Method Component.

In method_component_extended structure, the `start_offset` item and `end_offset` are byte offsets into the method_component_block containing this exception_handler_info structure in the Method Component.

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 the same method_info structure to an opcode of an instruction, or must be equal to the method’s bytecode count, the length of the bytecodes array of the method_info structure. The value of the `start_offset` must be less than the value of the `end_offset`. 
The \texttt{start\_offset} is \textit{inclusive} and the \texttt{end\_offset} is \textit{exclusive}; that is, the exception handler must be active while the execution address is within the interval \((\texttt{start\_offset}, \texttt{end\_offset})\).

\textbf{stop\_bit}

The \texttt{stop\_bit} item indicates whether the active range (try block) of this exception handler is contained within or is equal to the active range of any succeeding \texttt{exception\_handler\_info} structures in this \texttt{exception\_handlers} array. At the Java source level, this indicates whether an active range is nested within another, or has at least one succeeding exception handler associated with the same range. The latter occurs when there is at least one succeeding catch block or a finally block.

The \texttt{stop\_bit} item is equal to 1 if the active range does not intersect with a succeeding exception handler’s active range, and this exception handler is the last handler applicable to the active range. It is equal to 0 if the active range is contained within the active range of another exception handler, or there is at least one succeeding handler applicable to the same active range.

The \texttt{stop\_bit} provides an optimization to be used during the interpretation of the \texttt{athrow} bytecode. As the interpreter searches for an appropriate exception handler, it may terminate the search of the exception handlers in this Method Component under the following conditions:

- the location of the current program counter is less than the \texttt{end\_offset} of this exception handler, and
- the \texttt{stop\_bit} of this exception handler is equal to 1.

When these conditions are satisfied it is guaranteed that none of the succeeding exception handlers in this Method Component will contain an active range appropriate for the current exception.

In 6.10.2 Exception Handler Example 6.10.2, the \texttt{stop\_bit} item is set for both the third and fourth handlers.

\textbf{handler\_offset}

The \texttt{handler\_offset} indicates the start of the exception handler.

In \texttt{method\_component\_compact}, the \texttt{handler\_offset} item represents a byte offset into the \texttt{info} item of the Method Component.

In \texttt{method\_component\_extended}, the \texttt{handler\_offset} item represents a byte offset into the \texttt{method\_component\_block} containing this \texttt{exception\_handler\_info} structure in the Method Component.

At the Java source level, this is equivalent to the beginning of a catch or finally block. The value of the item must be a valid offset into a bytescodes array of a \texttt{method\_info} structure to an opcode of an instruction, and must be less than the value of the method’s bytecode count.

\textbf{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 (6.8 Constant Pool Component). 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.

The order of constants in the constant pool is constrained such that all entries referenced by catch_type_index items that represent catch block (not finally blocks) are located at non-zero entries.

6.10.4 method_info

The method_info structure is defined as follows:

method_info {
    method_header_info method_header
    ul 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 {
    ul bitfield {
        bit[4] flags
        bit[4] max_stack
    }
    ul bitfield {
        bit[4] nargs
        bit[4] max当地人
    }
}

extended_method_header_info {
    ul bitfield {
        bit[4] flags
        bit[4] padding
    }
    ul max_stack
    ul nargs
    ul max当地人
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.

**Table 6-12: CAP File Method Flags**

<table>
<thead>
<tr>
<th>Flags</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_EXTENDED</td>
<td>0x8</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x4</td>
</tr>
</tbody>
</table>

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.

All other flag values are reserved. 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 words required on the operand stack during execution of this method.

Stack entries of type `int` are represented in two words, while all others are represented in one word. See 3.2 Words

**nargs**

The `nargs` item indicates the number of words 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 words, while all others are represented in one word. See 3.2 Words.

**max_locals**
The `max_locals` item indicates the number of words required to represent the local variables declared by this method, not including the parameters passed to this method on invocation.\(^\text{10}\)

Local variables of type `int` are represented in two words, while all others are represented in one word (3.2 Words). If an entry in the local variables array of the stack frame is reused to store more than one local variable (for example, local variables from separate scopes), the number of words required for storage is two if one or more of the local 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 cannot be present in the `bytecodes` array item.

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

### 6.11 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 CAP file, referred to as the static field image. Offsets to particular static fields are offsets into the static field image, not the Static Field Component.

Final static fields of primitive types are not represented in the static field image. Instead these compile-time constants must be placed in line in Java Card technology-based instructions (“Java Card instructions”).

The Static Field Component includes all information required to initialize classes. In the Java virtual machine a class is initialized by executing its `<clinit>` method. In the Java Card virtual machine the functionality of `<clinit>` methods is represented in the Static Field Component as array initialization data and non-default values of primitive types data. 2.2.4.6 Limitations of Class Initialization contains a description of the subset of `<clinit>` functionality supported in the Java Card virtual machine.

The Static Field Component does not reference any other component in this CAP file. The Constant Pool Component (6.8 Constant Pool Component), Export Component (6.13 Export Component), Descriptor Component (6.14 Descriptor Component), and Debug Component (6.15 Debug Component) reference fields in the static field image defined by the Static Field Component.

The ordering constraints, or segments, associated with a static field image are shown in Table 6-13. 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.

\(^{10}\) 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.
Table 6-13: Segments of a Static Field Image

<table>
<thead>
<tr>
<th>Category</th>
<th>Segment</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference types</td>
<td>1</td>
<td>arrays of primitive types initialized by &lt;clinit&gt; methods</td>
</tr>
<tr>
<td>reference types</td>
<td>2</td>
<td>reference types initialized to null, including arrays</td>
</tr>
<tr>
<td>primitive types</td>
<td>3</td>
<td>primitive types initialized to default values</td>
</tr>
<tr>
<td>primitive types</td>
<td>4</td>
<td>primitive types initialized to non-default values</td>
</tr>
</tbody>
</table>

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

Table 6-14: Static Field Sizes

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>1</td>
</tr>
<tr>
<td>Byte</td>
<td>1</td>
</tr>
<tr>
<td>Short</td>
<td>2</td>
</tr>
<tr>
<td>Int</td>
<td>4</td>
</tr>
<tr>
<td>reference, including arrays</td>
<td>2</td>
</tr>
</tbody>
</table>

The `static_field_component` structure is defined as:

```java
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 CAP file, 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 6-14.

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

The value of the `image_size` is defined as:

```java
image_size =
reference_count * 2 +
default_value_count +
non_default_value_count.
```

The `reference_count` item indicates the number of reference type static fields defined in this CAP file. This is the number of fields represented in segments 1 and 2 of the static field image as described in Table 6-13.

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

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 6-13. It represents the number of arrays initialized in all of the `<clinit>` methods in this CAP file.

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

The `array_init[]` 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:

```java
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.
Table 6-15: Array Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>2</td>
</tr>
<tr>
<td>Byte</td>
<td>3</td>
</tr>
<tr>
<td>short</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>5</td>
</tr>
</tbody>
</table>

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 the Java programming language), since the values array is an array of bytes and the static field array may be a non-byte type. The Java programming language 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 6-14) 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 6-14.

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 6-13. 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 6-14.

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 6-13. 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 6-13. 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 6-14.

The value of a boolean type is 1 to represent true and 0 to represent false.
6.12 Reference Location Component

The Reference Location Component represents lists of offsets into the info item of the Method Component (6.10 Method Component) to items that contain indices into the constant_pool[] array of the Constant Pool Component (6.8 Constant Pool Component). This includes all constant pool index operands of instructions, and all non-zero catch_type_index items of the exception_handlers array. The catch_type_index items that have the value of 0 are not included since they represent finally blocks instead of particular exception classes.

Some of the constant pool indices are represented in one-byte values while others are represented in two-byte values. Operands of getfield_T and putfield_T instructions are one-byte constant pool indices. All other indices in a Method Component are 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 one of the following structures based on Compact or Extended format of the CAP file:

reference_location_component_compact {
    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]
}

reference_location_component_extended { (since CAP format 2.3)
    u1 tag
    u4 size
    u1 reference_location_component_block_count
    reference_location_component_block blocks[]
}

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.

reference_location_component_block_count

The reference_location_component_block_count item in reference_location_component_extended structure indicates the number of reference_location_component_block entries in the blocks array. The value of reference_location_component_block_count item must be equal to the method_component_block_count item in Method Component.

blocks[

The blocks[] item of the reference_location_component_extended structure represents an array of reference_location_component_block items. The number of entries in this array must be the same as the value of reference_location_component_block_count.

6.12.1 reference_location_component_block
A reference_location_component_extended may contain between 1 and 127 reference_location_component_block items. Each block can have a maximum size of 65535 bytes. Each block corresponds to method_component_block items in the blocks array in the Method Component i.e. offsets in a reference_location_component_block at index i in the blocks array in the Reference Location Component must be for the method_component_block item in the blocks array at the same index i in the Method Component.

The reference_location_component_block structure is defined as follows:

reference_location_component_block {
  u2 byte_index_count
  u1 offsets_to_byte_indices[byte_index_count]
  u2 byte2_index_count
  u1 offsets_to_byte2_indices[byte2_index_count]
}

byte_index_count

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

offsets_to_byte_indices[]

In reference_location_component_compact structure, 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.
In reference_location_component_block in reference_location_component_compact structure, the offsets_to_byte_indices item represents an array of 1-byte jump offsets into the corresponding method_component_block in the Method Component's blocks array 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 \( n \text{th} \) 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.

Table 6-16: One-byte Reference Location Example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Offset to Operand</th>
<th>Jump Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>getfield_a 0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>putfield_b 2</td>
<td>65</td>
<td>55</td>
</tr>
<tr>
<td>getfield_s 1</td>
<td>580</td>
<td>255, 255, 5</td>
</tr>
<tr>
<td>putfield_a 0</td>
<td>835</td>
<td>255, 0</td>
</tr>
<tr>
<td>getfield_i 3</td>
<td>843</td>
<td>8</td>
</tr>
</tbody>
</table>

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[]

In reference_location_component_compact structure, the offsets_to_byte2_indices item represents an array of 1-byte jump offsets into the info item of the Method Component to each 2-byte constant_pool[] array index.

In reference_location_component_block in reference_location_component_compact structure, the offsets_to_byte2_indices item represents an array of 1-byte jump offsets into the corresponding method_component_block in the Method Component's blocks array 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 \( n \text{th} \) 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 6-16. 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.

### 6.13 Export Component

The Export Component lists all static elements in this CAP file that may be imported by classes in other packages. Instance fields and virtual methods are not represented in the Export Component. For Extended format of CAP files, a CAP file may contain private packages. Export component must not contain any information from these packages. Packages in the Extended format CAP files must be in the same order as they are in the Header Component (6.4 Header Component).

For package represented by Compact format CAP, if the CAP file contains the export component, the package is considered a public package.

For public packages that include applets, the Export Component includes entries only for all public interfaces that are shareable.\(^\text{11}\) For public packages that do not include any applets, the Export Component contains an entry for each public class and public interface. 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.

An interface is shareable 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 (6.9 Class Component), Method Component (6.10 Method Component), and Static Field Component (6.11 Static Field Component). No other component in this CAP file references the Export Component.

The Export Component is represented by one of the following structures based on Compact or Extended format of the CAP file:

```plaintext
export_component_compact {
  u1 tag
  u2 size
  u1 class_count
  class_export_info {
    u2 class_offset
  }
```

\(^{11}\) The restriction on shareable functionality is imposed by the firewall as defined in the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*. 

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]
}

export_component_extended { (since CAP format 2.3)
  u1 tag
  u2 size
  u1 package_count
  package_export_info{
    u1 class_count
    class_export_info {
      u2 class_offset
      u1 static_field_count
      u1 static_method_count
      u2 static_field_offsets[static_field_count]
      method_block_info static_methods[static_method_count]
    } class_exports[class_count]
  }package_exports[package_count]
}

The items of these structures 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.

package_count

The package_count item in export_component_extended structure represents the number of entries in the package_exports table. The value of the package_count item must be greater than 0.

package_exports[]

The package_exports item in export_component_extended structure represents a variable-length table of package_export_info structures for all public packages in this CAP file.
The items in `package_export_info` structure are:

- `class_count`

The `class_count` item in `package_export_info` structure and in `export_component_compact` structure represents the number of entries in the `class_exports` table. The value of the `class_count` item must be greater than zero.

`class_exports[]`

The `class_exports` item represents a variable-length table of `class_export_info` structures. For library CAP files, the table contains an entry for each of the public classes and public interfaces defined in this CAP file. For application CAP files, the table contains an entry for each of the public shareable interfaces defined in this CAP file.

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 Classes and Interfaces). The token value is published in the Export file (5.7 Classes and Interfaces) of the package containing the class.

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 (6.9 Class Component). For library CAP files, the item at that offset must be either an `interface_info` or a `class_info` structure. The `interface_info` or `class_info` structure at that offset must represent the exported class or interface.

For application CAP files, 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_SHARABLE 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 (6.11 Static Field Component). 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 (5.9 Methods) 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 (6.10 Method Component). 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.

**static_methods**

The **static_methods** item is an array of **method_block_info** (6.9.2.4 **method_block_info**) structures. The **method_info** structure pointed to by the **method_block_info** structure must represent the exported static method or constructor.

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

**6.14 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 (6.8 Constant Pool Component), Class Component (6.9 Class Component), Method Component (6.10 Method Component), and Static Field Component (6.11 Static Field Component). No components in the CAP file reference the Descriptor Component.

Descriptor component in the Extended format contains information about all public and private packages contained in the CAP. Public packages in the CAP file must be described first and must be in
the same order as they are in the Header Component (6.4 Header Component) followed by private packages.

The Descriptor Component is represented by one of the following structures based on Compact or Extended format of the CAP file:

```java
descriptor_component_compact {
    u1 tag
    u2 size
    u1 class_count
    class_descriptor_info_compact classes[class_count] (since CAP format 2.3)
    type_descriptor_info types
}

descriptor_component_extended { (since CAP format 2.3)
    u1 tag
    u4 size
    u1 package_count
    package_descriptor_info packages[package_count]
    type_descriptor_info types
}
```

The items of these structures are as follows:

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.

package_count

The package_count item in `descriptor_component_extended` structure represents the number of entries in the packages table.

packages[]

The packages item represents a table of variable-length `package_descriptor_info` structures. Each package defined in this CAP file is represented in the table.
6.14.1 package_descriptor_info
The package_descriptor_info structure describes a package defined in this CAP file.

```plaintext
package_descriptor_info { (since CAP format 2.3)
    u1 class_count
    class_descriptor_info_extended classes[class_count]
}
```

The items of these structures are as follows:

class_count

The class_count item represents the number of entries in the classes table in descriptor_component_compact and package_descriptor_info structures.

classes[]

The classes item represents a table of variable-length class_descriptor_info_compact table in descriptor_component_compact structures or class_descriptor_info_extended in package_descriptor_info structure. 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 CAP file. Those referenced are enumerated in the Constant Pool Component.

6.14.2 class_descriptor_info_compact and class_descriptor_info_extended
The class_descriptor_info_compact and class_descriptor_info_extended structures are used to describe a class or interface defined in this package:

```plaintext
class_descriptor_info_compact { (since CAP format 2.3)
    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_compact methods[method_count]
}
```

class_descriptor_info_extended { (since CAP format 2.3)
    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_extended methods[method_count]
}

The items of these structures are as follows:

token

The token item represents the class token (4.3.7.2 Classes and Interfaces) 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.

Table 6-17: CAP File Class Descriptor Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x01</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x10</td>
</tr>
<tr>
<td>ACC_INTERFACE</td>
<td>0x40</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x80</td>
</tr>
</tbody>
</table>

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.

All other flag values are reserved. 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 (6.9 Class Component). The class_ref structure is defined as part of the CONSTANT_Classref_info structure (6.8.1 CONSTANT_Classref).

interface_count
The `interface_count` item represents the number of entries in the interfaces array. For an interface, `interface_count` is always set to zero.

`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.

Static final fields of primitive types are not represented as fields in a CAP file, but instead these compile-time constants are placed inline in bytecode sequences. The `field_count` item does not include static final field of primitive types defined by this class.

`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. The elements in the array are `class_ref` structures indicating the location of the `interface_info` structure in the Class Component (6.9 Class Component). The `class_ref` structure is defined as part of the `CONSTANT_Classref_info` structure (6.8.1 CONSTANT_Classref).

`fields[]`

The `fields` item represents an array of `field_descriptor_info` structures. Each field declared by this class is represented in the array, except static final fields of primitive types. Inherited fields are not included 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. For a class, inherited methods are not included in the array. For an interface, inherited methods are included in the array.

### 6.14.3 `field_descriptor_info`

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

```java
field_descriptor_info {
  u1 token
  u1 access_flags
  union {
    static_field_ref static_field
  }
  class_ref class
  u1 token
}```
The items of the `field_descriptor_info` structure are 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x01</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x02</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x04</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x08</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x10</td>
</tr>
</tbody>
</table>

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.

All other flag values are reserved. 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 (6.8.3 CONSTANT_StaticFieldref and CONSTANT_StaticMethodref).

  If the ACC_STATIC flag is equal to 0, this item represents a reference to an instance field. It contains a class_ref item and an instance field token item. These items are defined in the same manner as in the `CONSTANT_InstanceFieldref` structure (6.8.2 CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref).
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.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>0x0002</td>
</tr>
<tr>
<td>Byte</td>
<td>0x0003</td>
</tr>
<tr>
<td>Short</td>
<td>0x0004</td>
</tr>
<tr>
<td>Int</td>
<td>0x0005</td>
</tr>
</tbody>
</table>

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.

6.14.4 method_descriptor_info_compact and method_descriptor_info_extended
The method_descriptor_info_compact and method_descriptor_info_extended structures are used to describe a method defined in this CAP file. This structure contains sufficient information to locate and parse the methods in the Method Component, while the Method Component does not.

```java
method_descriptor_info_compact {
    u1 token
    u1 access_flags
    u2 method_offset
    u2 type_offset
    u2 bytecode_count
    u2 exception_handler_count
    u2 exception_handler_index
}
```
methodDescriptorInfoExtended { (since CAP format 2.3)
    u1 token
    u1 access_flags
    u1 methodComponentBlockIndex
    u2 method_offset
    u2 type_offset
    u2 bytecode_count
    u2 exception_handler_count
    u2 exception_handler_index
}

The items of these structure are as follows:

The items of the methodDescriptorInfo structure are as follows:

token

The token item represents the static method token (4.3.7.4 Static Methods and Constructors) or virtual method token (4.3.7.6 Virtual Methods) or interface method token (4.3.7.7 Interface Methods) 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.

Table 6-20: CAP File Method Descriptor Flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x01</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x02</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x04</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x08</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x10</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x40</td>
</tr>
<tr>
<td>ACC_INIT</td>
<td>0x80</td>
</tr>
</tbody>
</table>

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 classfile. 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 method. In Java a constructor method is recognized by its name, <init>, but in Java Card systems, the name is replaced
by a token. As in the Java verifier, these methods require special checks by the verifier for the Java Card platform ("Java Card verifier").

All other flag values are reserved. Their values must be zero.

**method_component_block_index**

The `method_component_block_index` item in `method_descriptor_info_extended`, represents the index into the blocks array of Method Component (6.10 Method Component). The `method_component_block` at that index contains the referenced method. If the `class_descriptor_info` structure that contains this `method_component_block_index` structure represents an interface, the value of the `method_component_block_index` item must be zero.

**method_offset**

For `descriptor_component_compact`, 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 (6.10 Method Component). The element at that offset must be the beginning of a `method_info` structure. The `method_info` structure must represent this method.

For `descriptor_component_extended`, 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 `method_component_block` in the blocks array of Method Component (6.10 Method Component) at index represented by `method_component_block_index`. 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_compact` or `class_descriptor_info_extended` 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 (6.10 Method Component) of this method.

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

`exception_handler_index`

For `descriptor_component_compact`, the `exception_handler_index` item represents the index to the first `exception_handlers` table entry in the method component (6.10 Method Component) 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.

For `descriptor_component_extended`, the `exception_handler_index` item represents the index to the first `exception_handlers` table entry in the `method_component_block` at index `method_component_block_index` in the `blocks` array in method component (6.10 Method Component) 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.

**6.14.5 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]
    type_descriptor 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 (6.8 Constant Pool Component).

`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. For a description of the `type_descriptor` structure, see 6.9.1 `type_descriptor`.

### 6.15 Debug Component

This section specifies the format for the Debug Component. The Debug Component contains all the metadata necessary for debugging packages contained in a CAP file on a suitably instrumented Java Card virtual machine. It is not required for executing Java Card programs in a non-debug environment.

The Debug Component references the Class Component (6.9 Class Component), Method Component (6.10 Method Component), and Static Field Component (6.11 Static Field Component). No components reference the Debug Component.

Debug component in the Extended format contains debug information about all public and private packages contained in the CAP. The order in which the packages are listed in the Debug Component must be the same as the order in the extended Descriptor Component (6.14 Descriptor Component).

The Debug Component is represented by one of the following structures based on Extended or Compact format of the CAP file:

```plaintext
debug_component_compact { (since CAP format 2.2)
  u1 tag
  u2 size
  u2 string_count
  utf8_info strings_table[string_count]
  package_debug_info_compact package
}
```
debug_component_extended { (since CAP format 2.3)
    u1 tag
    u4 size
    u2 string_count
    utf8_info strings_table[string_count]
    u1 package_count
    package_debug_info_extended packages[package_count]
}

The items in these structures are defined as follows:

tag

The tag item has the value COMPONENT_Debug (12).

size

The number of bytes in the component, excluding the tag and size items. The value of size must be greater than zero.

string_count

The number of strings in the strings_table[] table.

strings_table[]

A table of all the strings used in this component. Various items that occur through this component represent unsigned two-byte indices into this table.

Each entry in the table is a utf8_info structure. A utf8_info structure is represented by the following structure:

utf8_info {
    u2 length
    u1 bytes[length]
}

The items in the utf8_info structure are defined as follows:

length

The number of bytes in the string.

bytes

The bytes of the string in UTF-8 format.
package_count

The package_count item in debug_component_extended structure represents the number of entries in the packages array. The value of the package_count item must be greater than 0.

6.15.1 package_debug_info_compact and package_debug_info_extended Structures

The package_debug_info_compact and package_debug_info_extended structures contain debug information for a single package. These structures are represented in the following formats:

package_debug_info_compact{
    u2 package_name_index
    u2 class_count
    class_debug_info_compact classes[class_count]
}

package_debug_info_extended{ (since CAP format 2.3)
    u2 package_name_index
    u2 class_count
    class_debug_info_extended classes[class_count]
}

package_name_index

Contains an index into the strings_table[] item. The strings_table[] item entry referenced by this index must contain the fully-qualified name of the package in this CAP file represented by this package_debug_info_compact or package_debug_info_extended structure.

class_count

The number of classes in the classes table.

classes[]

Contains a single class_debug_info_compact or class_debug_info_extended structure in package_debug_info_compact or package_debug_info_extended structures respectively, for each class in this package.

6.15.2 The class_debug_info_compact and class_debug_info_extended Structures

The class_debug_info_compact and class_debug_info_extended structures are for CAP files in Compact and Extended formats respectively and contain all of the debugging information for a class or interface. They also contain tables of debugging information for all the classes’ fields and methods.
class_debug_info_compact {
    u2 name_index
    u2 access_flags
    u2 location
    u2 superclass_name_index
    u2 source_file_index
    u1 interface_count
    u2 field_count
    u2 method_count
    u2 interface_names_indexes[interface_count]
    field_debug_info fields[field_count]
    method_debug_info_compact methods[method_count]
}

class_debug_info_extended { (since CAP format 2.3)
    u2 name_index
    u2 access_flags
    u2 location
    u2 superclass_name_index
    u2 source_file_index
    u1 interface_count
    u2 field_count
    u2 method_count
    u2 interface_names_indexes[interface_count]
    field_debug_info fields[field_count]
    method_debug_info_extended methods[method_count]
}

The items in these structures are defined as follows:

name_index

Contains an index into the strings_table[] item of the debug_component structure. The
strings_table[] entry at the indexed location must be the fully-qualified name of this class.

access_flags

A two-byte mask of modifiers that apply to this class. The modifiers are:

Table 6-21: Class Access and Modifier Flags

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
</tr>
</tbody>
</table>
The **ACC_SHAREABLE** flag indicates whether this class or interface is shareable.\(^\text{12}\) A class is shareable if it implements (directly or indirectly) the `javacard.framework.Shareable` interface. An interface is shareable if it is or extends (directly or indirectly) the `javacard.framework.Shareable` interface.

The **ACC_REMOTE** flag indicates whether this class or interface is remote. The value of this flag must be one if and only if the class or interface satisfies the requirements defined in 2.2.6.1 Remote Classes and Remote Interfaces.

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*.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_REMOTE</td>
<td>0x0020</td>
</tr>
<tr>
<td>ACC_INTERFACE</td>
<td>0x0200</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
</tr>
<tr>
<td>ACC_SHAREABLE</td>
<td>0x0800</td>
</tr>
</tbody>
</table>

\(^\text{12}\) The **ACC_SHAREABLE** flag is defined to enable Java Card virtual machines to implement the firewall restrictions defined by the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*. 

The byte offset of the class_info or interface_info record for this class or interface into the info item of the Class Component (6.9 Class Component).

**superclass_name_index**

Contains an index into the strings_table[] item of the debug_component structure. The strings_table[] entry at the indexed location must be the fully-qualified name of the superclass of this class or the string “null” if the class has no superclass.

**source_file_index**

Contains the index into the strings_table[] item of the debug_component structure. The strings_table[] entry at the indexed location must be the name of the source file in which this class is defined.

**interface_count**

The number of indexes in the interface_names_indexes[] table.

**field_count**

The number of field_debug_info structures in the fields[] table.
method_count

The number of method_debug_info structures in the methods[] table.

interface_names_indexes[]

Contains the indexes into the strings_table[] item of the debug_component structure. The strings_table[] entry at each indexed location must be the name of an interface implemented by this class. There must be an index value present for every interface implemented by this class, including interfaces implemented by superclasses of this class and superinterfaces of the implemented interfaces.

If ACC_INTERFACE is set, the strings_table[] entry at each indexed location must be the name of a super interface directly or indirectly extended by this interface. There must be an index value present for every super interface directly or indirectly extended by this interface.

fields[]

Contains field_debug_info structures for all the fields declared by this class, including static final fields of primitive types. Inherited fields are not included in this array.

methods[]

Contains method_debug_info_compact or method_debug_info_extended structures in class_debug_info_compact or class_debug_info_extended structures respectively, for all the methods declared or defined in this class. Inherited methods are not included in this array.

6.15.2.1 The field_debug_info Structure

The field_debug_info structure describes a field in a class. It can describe either an instance field, a static field, or a constant (primitive final static) field. The contents union will have the form of a token_var if the field is an instance field, a location_var if it is a static field, or a const_value if it is a constant.

The field_debug_info structure is defined as follows:

field_debug_info {
    u2 name_index
    u2 descriptor_index
    u2 access_flags
    union {
        
        u1 pad1
        u1 pad2
        u1 pad3
        u1 token
    } token_var
The items in the field_debug_info structure are defined as follows:

name_index

Contains an index into the strings_table[] item of the debug_component structure. The strings_table[] entry at the indexed location must be the simple (meaning, not fully-qualified) name of the field (for example, “applets”).

descriptor_index

Contains an index into the strings_table[] item of the debug_component structure. The strings_table[] entry at the indexed location must be the type of the field. Class types are fully-qualified (for example, “[Ljavacard/framework/Applet;”).

access_flags

A two-byte mask of modifiers that apply to this field.

Table 6-22: Field Access and Modifier Flags

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
</tr>
</tbody>
</table>

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

contents

A field_debug_info structure can describe an instance field, a static field, or a static final field (a constant). Constants can be either primitive data or arrays of primitive data. Depending on the kind of field described, the contents item is interpreted in different ways. The kind and type of the field can be determined by examining the field’s descriptor and access flags.

token_var
If the field is an instance field, this value is the instance field token of the field. The `pad1`, `pad2`, and `pad3` items are padding only; their values should be ignored.

`location_var`

If the field is a non-final static field or a final static field with an array type (a constant array), this value is the byte offset of the location for this field in the static field image defined by the Static Field Component (6.11 Static Field Component). The pad item is padding only; its value should be ignored.

`const_value`

If the field is a final static field of type `byte, boolean, short, or int`, this value is interpreted as a signed 32-bit constant.

**6.15.2.2 The method_debug_info_compact and method_debug_info_extended Structures**

The `method_debug_info_compact` and `method_debug_info_extended` structures describe a method of a class. They can describe methods that are either virtual or non-virtual (static or initialization methods). The structures are defined as follows:

```java
method_debug_info_compact {
    u2 name_index
    u2 descriptor_index
    u2 access_flags
    u2 location
    u1 header_size
    u2 body_size
    u2 variable_count
    u2 line_count
    variable_info variable_table[variable_count]
    line_info line_table[line_count]
}
```

```java
method_debug_info_extended { (since CAP format 2.3)
    u2 name_index
    u2 descriptor_index
    u2 access_flags
    u1 method_component_block_index
    u2 location
    u1 header_size
    u2 body_size
    u2 variable_count
    u2 line_count
    variable_info variable_table[variable_count]
    line_info line_table[line_count]
}
```
The items in these structures are defined as follows:

**name_index**
Contains an index into the `strings_table[]` item of the `debug_component` structure. The `strings_table[]` entry at the indexed location must be the simple (meaning, not fully-qualified) name of the method (for example, “lookupAID”).

**descriptor_index**
Contains an index into the `strings_table[]` item of the `debug_component` structure. The `strings_table[]` entry at the indexed location must be the argument and return types of the method (meaning, the signature without the method name). Class types are fully-qualified (for example, “([BSB)Ljavacard/framework/AID;”)  

**access_flags**
A two-byte mask of modifiers that apply to this method.

### Table 6-23: Method Modifier Flags

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
</tr>
<tr>
<td>ACC_NATIVE</td>
<td>0x0100</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
</tr>
</tbody>
</table>

The **ACC_NATIVE** flag is only valid for methods of a package located in the card mask. It cannot be used for methods contained in a CAP file.

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

**method_component_block_index**

The **method_component_block_index** item in `method_debug_info_extended` represents the index into the blocks array of Method Component (6.10 Method Component). The **method_component_block** at that index contains the referenced method. Abstract methods may have value zero for **method_component_block_index** or have value representing the index into the blocks array of the Method Component (6.10 Method Component) that has the corresponding **method_info** structute with **ACC_ABSTRACT** flag set.
Abstract methods may have zero as value for method_component_block_index or have a value corresponding to the method component block containing the method_info structure with ACC_ABSTRACT flag set.

location

In method_debug_info_compact, location is a byte offset of the method_info structure for this method into the info item of the Method Component (6.10 Method Component).

In method_debug_info_extended, location is a byte offset into the method_component_block in the blocks array of Method Component (6.10 Method Component) at index represented by method_component_block_index. The method_info structure at offset must represent the referenced method.

Abstract methods may have a location of zero or have a valid offset to the corresponding method_info structure with ACC_ABSTRACT flag set.

header_size

The size in bytes of the header of the method. Abstract methods have a header_size of zero.

body_size

The size in bytes of the body of the method, not including the method header. Abstract methods have a body_size of zero.

variable_count

The number of variable_info entries in the variable_table[] item. Abstract methods have a variable_count of zero.

line_count

The number of line_info entries in the line_table[] item. Abstract methods have a line_count of zero.

variable_table[]

Contains the variable_info structures for all variables in this method.

The variable_info structure describes a single local variable of a method. It indicates the index into the local variables of the current frame at which the local variable can be found, as well as the name and type of the variable. It also indicates the range of bytecodes within which the variable has a value.

variable_info {
    u1 index
    u2 name_index
    u2 descriptor_index
}
The items in the variable_info structure are defined as follows:

index

The index of the variable in the local stack frame, as used in load and store bytecodes. If the variable at index is of type int, it occupies both index and index + 1.

name_index

Contains an index into the strings_table[] item of the debug_component structure. The strings_table[] entry at the indexed location must be the name of the local variable, (for example, “applets”).

descriptor_index

Contains an index into the strings_table[] item of the debug_component structure. The strings_table[] entry at the indexed location must be the type of the local variable. Class types are fully-qualified (for example, “[Ljavacard/framework/Applet;”).

start_pc

The index of the first bytecode in which the variable is in-scope and valid.

length

Number of bytecodes in which the variable is in-scope and valid. The value of start_pc + length will be either the index of the next bytecode after the valid range, or the first index beyond the end of the bytecode array.

line_table[]

Contains the line_info structures that map bytecode instructions of this method to lines in the class’s source file.

Each line_info item represents a mapping of a range of bytecode instructions to a particular line in the source file that contains the method. The range of instructions is from start_pc to end_pc, inclusive. start_pc and end_pc represent a zero-based byte offset within the method. The source_line is the one-based line number in the source file. The structure is defined as follows:

line_info {
    u2 start_pc
    u2 end_pc
    u2 source_line
}
The items in the `line_info` structure are defined as follows:

`start_pc`

The byte offset of the first bytecode in the range of instructions.

`end_pc`

The byte offset of the last operand of the last bytecode in the range of instructions.

`source_line`

Line number in the source file.

### 6.16 Static Resource Component

This section specifies the format for the Static Resource Component. Static Resource Component must be present if any package in this CAP file has any static resources. If none of the packages in this CAP file has any static resources, this component must not be present in this CAP file.

The Static Resource Component may contain any static resource that can be represented in a byte format. Size of each static resource must be between 0 and 32767 bytes.

The Static Resource Component does not reference any other component.

Static Resource Component must always be represented in the Extended Format as specified in these specifications regardless of compact or Extended format of the CAP file. The Static Resource Component is represented by the following structure:

```
static_resource_component {  
    u1 tag  
    u4 size  
    u2 resource_count  
    resource_directory_info resource_directory[resource_count]  
    static_resource_info static_resources[resource_count]  
}
```

The items in this structure are defined as follows:

`tag`

The tag item has the value `COMPONENT_Static_resource` (13).

`size`

The number of bytes in the component, excluding the `tag` and `size` items. The value of `size` must be greater than zero.
resource_count

The number of resources represented in this component. Value for resource_count cannot be 0. This number represents the number of entries in the directory and static_resources arrays.

resource_directory []

Contains the resource_directory_info structures for all static resources represented in this component. The number of entries in this table must be equal to resource_count.

The resource_directory_info structure describes a single directory entry in the directory table. It indicates the id and the size of each resource represented in this component.

resource_directory_info {
  u2 resource_id
  u4 resource_size
}

The items in the resource_directory_info structure are described below.

resource_id

This item represents the id of a resource in this component. The value for resource_id must be unique in a CAP file.

resource_size

The resource_size item represents the size in bytes of the resource represented by resource_id. Value of this item must be between 0 and 32767 bytes.

static_resources[]

Contains the static_resource_info structures for all static resources represented in this component. The number of entries in this table must be equal to resource_count.

The static_resource structure describes a single entry in the static_resources table. This structure is represented as follows.

static_resource_info {
  ul static_resource[resource_size]
}

The static_resource item in the static_resource_info structure is described below.

static_resource
This item is an array of bytes representing a static resource. The size of the array must be equal to resource_size entry in the corresponding resource_directory_info entry in the resource_directory.
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 CA P 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 CA P 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 CA P File Format), two opcodes are reserved for internal use by a Java Card virtual machine implementation. If Oracle 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 Virtual Machine Specification, Classic Edition 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 *Virtual Machine Specification, Classic Edition* 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 Contexts) in which the instruction is executing is different than the owning context (3.4 Contexts) 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 *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*. An important detail to note is that any cross-context method invocation will result in a context switch (3.4 Contexts).

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 *Virtual Machine Specification, Java Card Platform, v3.1, Classic Edition* 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 in alphabetical order by entries of the form shown in Table 7-1, an example instruction entry.

**Table 7-1: Example Instruction Entry**

<table>
<thead>
<tr>
<th>Example Instruction Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>mnemonic</td>
</tr>
<tr>
<td>Short description of the instruction.</td>
</tr>
</tbody>
</table>

**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, and so on.

**Runtime Exception**

If any runtime exceptions can be thrown by the execution of 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.
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 Frames) of the current frame (3.5 Frames) 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 Virtual Machine Specification, Java Card Platform, v3.1, Classic Edition 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.

7.5.1 aaload
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.

Otherwise if the arrayref references a write-only array view, the aaload instruction throws a SecurityException.

Notes

In some circumstances, the aaload instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array or array view referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

7.5.2 *aastore*

Store into reference array or reference array view

Format

*aastore*

Forms

*aastore* = 55 (0x37)

Stack

..., arrayref, index, value
... arrayref, index, value -> ...

**Description**

The arrayref must be of type reference and must refer to an array or an array view, 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.

If the array referenced by arrayref is integrity-sensitive, its integrity is checked before the value is stored. The integrity control element is updated when the value is stored. The whole operation (value storage and the integrity control element update) is performed atomically.

At runtime the type of value must be confirmed to 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\(^\text{13}\), then:
  - If T is a class type, then T must be Object (2.2.1.4 Unsupported Classes);
  - 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, 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 Data Types and Values).
    - TC and SC are reference types\(^\text{14}\) (3.1 Data Types and Values) 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.

**Runtime Exceptions**

If arrayref is null, a astore throws a NullPointerException.

---

\(^{13}\) When both S and T are arrays of reference types, this algorithm is applied recursively using the types of the arrays, namely SC and TC. In the recursive call, S, which was SC in the original call, may be an interface type. This rule can only be reached in this manner. Similarly, in the recursive call, T, which was TC in the original call, may be an interface type.

\(^{14}\) This version of the Java Card virtual machine does not support multi-dimensional arrays. Therefore, neither SC or TC can be an array type.
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.

Otherwise if the array referenced by arrayref is integrity-sensitive and an inconsistency is detected during the array integrity check, the aastore instruction throws a SecurityException.

Otherwise if the arrayref references a read-only array view, the aastore instruction throws a SecurityException.

Notes

In some circumstances, the aastore instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.

7.5.3 aconst_null
Push null

Format
aconst_null

Forms
aconst_null = 1 (0x1)

Stack
... -> ..., null

Description
Push the null object reference onto the operand stack.

7.5.4 aload
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 Frames). 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.

7.5.5 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 Frames). 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.

7.5.6 anewarray
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 Frames), 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 (6.8.1 CONSTANT_Classref), 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.

7.5.7 areturn
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 Frames) 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.

### 7.5.8 arraylength

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 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.

### 7.5.9 astore

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 Frames). 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 aaload 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.

7.5.10 astore_<n>
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 Frames). 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 astore_<n> instructions is the same as astore with an index of <n>, except that the operand <n> is implicit.

7.5.11 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 Frames) 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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact
circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

### 7.5.12 baload

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.

Otherwise if the arrayref references a write-only array view, the baload instruction throws a SecurityException.

**Notes**

In some circumstances, the baload instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

### 7.5.13 bastore

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.

If the array referenced by arrayref is integrity-sensitive, its integrity is checked before the value is stored. The integrity control element is updated when the value is stored. The whole operation (value storage and the integrity control element update) is performed atomically.

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.

Otherwise if the array referenced by arrayref is integrity-sensitive and an inconsistency is detected during the array integrity check, the bastore instruction throws a SecurityException.

Otherwise if the arrayref references a read-only array view, the bastore instruction throws a SecurityException.

Notes

In some circumstances, the bastore instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.

7.5.14 bipush
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.

Note: If a virtual machine does not support the int data type, the bipush instruction will not be available.

7.5.15 bspush

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.

7.5.16 checkcast

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:

Table 7-2: Array Values

<table>
<thead>
<tr>
<th>Array Type</th>
<th>atype</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_BOOLEAN</td>
<td>10</td>
</tr>
<tr>
<td>T_BYTE</td>
<td>11</td>
</tr>
<tr>
<td>T_SHORT</td>
<td>12</td>
</tr>
<tr>
<td>T_INT</td>
<td>13</td>
</tr>
<tr>
<td>T_REFERENCE</td>
<td>14</td>
</tr>
</tbody>
</table>

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 Frames), 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 (6.8.1 CONSTANT_Classref), 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\(^\text{15}\), then:
  - If T is a class type, then T must be Object (2.2.1.4 Unsupported Classes);

\(^{15}\) When both S and T are arrays of reference types, this algorithm is applied recursively using the types of the arrays, namely SC and TC. In the recursive call, S, which was SC in the original call, may be an interface type. This rule can only be reached in this manner. Similarly, in the recursive call, T, which was TC in the original call, may be an interface type.
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, 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 Data Types and Values).
    - TC and SC are reference types\(^\text{16}\) (3.1 Data Types and Values) 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.

**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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the value of atype may not be 13 (array type = T_INT).

### 7.5.17 dup

Duplicate top operand stack word

**Format**

\[
\text{dup}
\]

**Forms**

\[
dup = 61 \text{ (0x3d)}
\]

**Stack**

\[
..., \text{word} \rightarrow ..., \text{word}, \text{word}
\]

---

\(^{16}\) This version of the Java Card virtual machine specification does not support multi-dimensional arrays. Therefore, neither SC or TC can be an array type.
**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.

### 7.5.18 dup_x

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.
7.5.19 dup2
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.

7.5.20 getfield_<t>
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 Frames). The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (6.8.2 `CONSTANT_InstanceFieldref`, `CONSTANT_VirtualMethodref`, `CONSTANT_SuperMethodref`), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then 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\(^\text{17}\). 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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the `Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition`.

If a virtual machine does not support the int data type, the getfield_i instruction will not be available.

### 7.5.21 getfield_<t>_this

Fetch field from current object

---

\(^{17}\) The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.
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 Frames). The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (6.8.2 `CONSTANT_InstanceFieldref`, `CONSTANT_VirtualMethodref`, `CONSTANT_SuperMethodref`), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then 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\(^\text{18}\). The value at that offset into the class instance referenced...

\(^{18}\) The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.
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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the `getfield_i_this` instruction will not be available.

**7.5.22 `getfield_<t>_w`**

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 Frames), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (6.8.2 `CONSTANT_InstanceFieldref`,...
CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref), a reference to a class and a field token. The item must resolve to a field of type reference.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then 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\(^\text{19}\). 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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the getfield_i_w instruction will not be available.

**7.5.23 getstatic_<t>**

Get static field from class

**Format**

```
getstatic_<t>  
indexbyte1  
indexbyte2  
```

**Forms**

\(^\text{19}\) The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.
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 Frames), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at the index must be of type CONSTANT\_StaticFieldref (6.8.3 CONSTANT\_StaticFieldref and CONSTANT\_StaticMethodref), a reference to a static field.

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.

### 7.5.24 goto

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.

7.5.25 goto_w
Branch always (wide index)

Format

\textit{goto\_w}
\begin{verbatim}
branchbyte1
branchbyte2
\end{verbatim}

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 \text{branchoffset} = (\text{branchbyte1} \ll 8) \mid \text{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.

7.5.26 i2b
Convert int to byte

Format

\textit{i2b}
\begin{verbatim}
i2b
\end{verbatim}

Forms

\text{i2b} = 93 (0x5d)

Stack

\text{..., 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.

7.5.27 i2s
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.

7.5.28 iadd
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 `iadd` 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.

7.5.29 `iaload`
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`. 
Otherwise if the arrayref references a write-only array view, the iaload instruction throws a SecurityException.

**Notes**

In some circumstances, the iaload instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the iaload instruction will not be available.

### 7.5.30 iand

**Boolean AND int**

**Format**

\[
\text{\textit{iand}}
\]

**Forms**

\[
iand = 84 (0x54)
\]

**Stack**

\[
..., \text{value1.word1, value1.word2, value2.word1, value2.word2} \rightarrow ..., \text{result.word1, result.word2}
\]

**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.

### 7.5.31 iastore

**Store into int array**

**Format**

\[
iastore
\]

**Forms**

\[
iastore = 58 (0x3a)
\]

**Stack**

\[
..., \text{value1.word1, value1.word2, value2.word1, value2.word2} \rightarrow ..., \text{result.word1, result.word2}
\]
..., 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.

If the array referenced by arrayref is integrity-sensitive, its integrity is checked before the value is stored. The integrity control element is updated when the value is stored. The whole operation (value storage and the integrity control element update) is performed atomically.

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.

Otherwise if the array referenced by arrayref is integrity-sensitive and an inconsistency is detected during the array integrity check, the iastore instruction throws a SecurityException.

Otherwise if the arrayref references a read-only array view, the iastore instruction throws a SecurityException.

Notes

In some circumstances, the iastore instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.

If a virtual machine does not support the int data type, the iastore instruction will not be available.

7.5.32 icmp

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.

7.5.33 icont_<_i>

Push int constant

Format

icont_<_i>

Forms

icont_m1 = 10 (0x09)
icont_0 = 11 (0xa)
icont_1 = 12 (0xb)
icont_2 = 13 (0xc)
icont_3 = 14 (0xd)
icont_4 = 15 (0xe)
icont_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 icont_<_i> instruction will not be available.

7.5.34 idiv

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 · 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.

**7.5.35 if_acmp<cond>**

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.

7.5.36 if_acmp<cond>_w
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.
7.5.37 if_scmp<cond>
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.

7.5.38 if_scmp<cond>_w
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.

7.5.39 if<cond>
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.

7.5.40 if<cond>_w
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.

7.5.41 ifnonnull
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.

7.5.42 ifnonnull_w
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.

7.5.43 ifnull
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.

7.5.44 ifnull_w
Branch if reference is null (wide index)

Format

\textit{ifnull}_w
branchbyte1
branchbyte2

Forms

\textit{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 \((\text{branchbyte1} \ll 8) \mid \text{branchbyte2}\). Execution proceeds at that offset from the address of the opcode of this \textit{ifnull}_{w} instruction. The target address must be that of an opcode of an instruction within the method that contains this \textit{ifnull}_{w} instruction.

Otherwise, execution proceeds at the address of the instruction following this \textit{ifnull}_{w} instruction.

7.5.45 iinc
Increment local int variable by constant

Format

\textit{iinc}
index
const

Forms

\textit{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 Frames). 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.

7.5.46 iinc_w
Increment local int variable by constant

Format

\[ \text{iinc_w} \]
\[ \text{index} \]
\[ \text{byte1} \]
\[ \text{byte2} \]

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 Frames). 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.

7.5.47 iipush
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.

**7.5.48 iload**

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 Frames). 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.

7.5.49 iload_<n>
Load int from local variable

Format

```
iload_<n>
```

Forms

```
iload_0 = 32 (0x20)
iload_1 = 33 (0x21)
iload_2 = 34 (0x22)
iload_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 Frames). 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.

7.5.50 ilookupswitch
Access jump table by key match and jump

Format

```
ilookupswitch
defaultbyte1
defaultbyte2
tpairs1
tpairs2
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.

7.5.51 imul
Multiply int

Format
\textit{imul}

\textbf{Forms}

\texttt{imul = 70 (0x46)}

\textbf{Stack}

\ldots, \texttt{value1.word1, value1.word2, value2.word1, value2.word2} -\rightarrow \ldots, \texttt{result.word1, result.word2}

\textbf{Description}

Both \texttt{value1} and \texttt{value2} must be of type \texttt{int}. The values are popped from the operand stack. The int result is \texttt{value1} * \texttt{value2}. The result is pushed onto the operand stack.

If an \textit{imul} instruction overflows, then the result is the low-order bits of the mathematical product as an \texttt{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.

\textbf{Notes}

If a virtual machine does not support the \texttt{int} data type, the \textit{imul} instruction will not be available.

\textbf{7.5.52 ineg}

\textbf{Negate int}

\textbf{Format}

\texttt{ineg}

\textbf{Forms}

\texttt{ineg = 76 (0x4c)}

\textbf{Stack}

\ldots, \texttt{value.word1, value.word2} -\rightarrow \ldots, \texttt{result.word1, result.word2}

\textbf{Description}

The value must be of type \texttt{int}. It is popped from the operand stack. The int result is the arithmetic negation of \texttt{value}, \texttt{-value}. The result is pushed onto the operand stack.

For \texttt{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 \texttt{int} results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all \texttt{int} values \texttt{x}, \texttt{-x} equals \texttt{(~x) + 1}.
**Notes**

If a virtual machine does not support the int data type, the ineg instruction will not be available.

7.5.53 instanceof
Determine if object is of given type

**Format**

\[
\text{instanceof} \\
\text{atype} \\
\text{indexbyte1} \\
\text{indexbyte2}
\]

**Forms**

\[
\text{instanceof} = 149 (0x95)
\]

**Stack**

\[
..., \text{objectref} \rightarrow ..., \text{result}
\]

**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:

<table>
<thead>
<tr>
<th>Array Type</th>
<th>atype</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_BOOLEAN</td>
<td>10</td>
</tr>
<tr>
<td>T_BYTE</td>
<td>11</td>
</tr>
<tr>
<td>T_SHORT</td>
<td>12</td>
</tr>
<tr>
<td>T_INT</td>
<td>13</td>
</tr>
<tr>
<td>T_REFERENCE</td>
<td>14</td>
</tr>
</tbody>
</table>

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 Frames), where the value of the index is (indexbyte1 \(\ll 8\)) | indexbyte2. The item at that index in the constant pool must be of type CONSTANT_Classref (6.8.1 CONSTANT_Classref), 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\(^\text{20}\), then:
  - If T is a class type, then T must be Object (2.2.1.4 Unsupported Classes);
  - 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, 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 Data Types and Values).
    - TC and SC are reference types\(^\text{21}\) (3.1 Data Types and Values) 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.

**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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the value of atype may not be 13 (array type = T_INT).

---

\(^{20}\) When both S and T are arrays of reference types, this algorithm is applied recursively using the types of the arrays, namely SC and TC. In the recursive call, S, which was SC in the original call, may be an interface type. This rule can only be reached in this manner. Similarly, in the recursive call, T, which was TC in the original call, may be an interface type.

\(^{21}\) This version of the Java Card virtual machine specification does not support multi-dimensional arrays. Therefore, neither SC or TC can be an array type.
7.5.54 invokeinterface
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 Frames), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at that index must be of type CONSTANT_Classref (6.8.1 CONSTANT_Classref), a reference to an interface class. The specified interface is resolved.

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 interface method must not be <init> or an instance initialization method.

The object-ref 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 (2.2.1.4 Unsupported Classes) 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 an 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 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 invokeinterface instruction throws a NullPointerException.

**Notes**

In some circumstances, the invokeinterface instruction may throw a SecurityException if the current context (3.4 Contexts) is not the context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*. If the current context is not the object’s context and the Java Card RE permits invocation of the method, the invokeinterface instruction will cause a context switch (3.4 Contexts) 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.

**7.5.54.1 Interface Method Resolution**

The resolution of the interface method is defined by the following algorithm. The virtual machine is not required to exactly perform the following steps provided the result is the same.

1. Verify access permission from current context to object O on the stack
2. Extract the interface I from the constant pool entry CONSTANT_Classref
3. Determine the dynamic type (class) C for object O on the stack
4. Search in the classes in the hierarchy of C for an implemented_interface_info structure for the interface I, i.e., starting with C1=C:
   a. Search the table interfaces of C1. If I is not found, then set C1 to the immediate super-class of C1 and recursively continue at step a
5. Extract the virtual token T1 of the method M in class C1 as T1=index[method] in the implemented_interface_info structure
6. Lookup the implementation of the method M in the hierarchy of C, i.e.:
   a. Determine a token T in class C that denotes the same method M as token T1 in class C1
   b. If the token T does not exist or if T is less than C.public_virtual_method_table_base or if C.public_virtual_method_table[T-C.public_virtual_method_table_base] is 0xFFFF, then recursively continue the lookup at step a, with C being now set to the super-class of C
   c. Otherwise, C.public_virtual_method_table[T-C.public_virtual_method_table_base] is the location in class C of the method M to be invoked
7. If no implemented_interface_info structure is found at step 4.a or if no method implementation has been found in the hierarchy or if a method is found but has the ACC_ABSTRACT flag, then the virtual machine must exit the interface method resolution and handle this as an error. See section 2.3.3.1.
7.5.55 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 Frames), 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 (6.8.3 CONSTANT_StaticFieldref and CONSTANT_StaticMethodref), 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 (6.8.2 CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref), 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 resolved method is protected, and it is a member of a superclass of the current class, and the method is not declared in the same package as the current class, then 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.

**7.5.56 invokestatic**
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 Frames), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The
constant pool item at that index must be of type `CONSTANT_StaticMethodref` (6.8.3
`CONSTANT_StaticFieldref` and `CONSTANT_StaticMethodref`), 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.

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.
7.5.56.1 Super Method Resolution

The resolution of the super method is defined by the following algorithm. The virtual machine is not required to exactly perform the following steps provided the result is the same.

1. Verify access permission from current context to object O on the stack
2. Extract the class C1 and token T1 from the constant pool entry CONSTANT_SuperMethodref
3. Lookup the method in the hierarchy starting with C being the direct super-class of C1:
   a. Determine a token T in class C that denotes the same method as token T1 in class C1
   b. If T is less than C.public_virtual_method_table_base or if C.public_virtual_method_table[T-
      C.public_virtual_method_table_base] is 0xFFFF, then recursively continue the lookup at step a, with C being now set to the super-class of C
   c. Otherwise, C.public_virtual_method_table[T-
      C.public_virtual_method_table_base] is the location in class C of the method to be invoked
4. if no method implementation has been found in the hierarchy, i.e., no token T exists or there are no more super-classes during the lookup, or if a method implementation has been found but has the ACC_ABSTRACT flag, then the virtual machine must exit the super method resolution and handle this as an error. See section 2.3.3.1.

7.5.57 invokevirtual

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 Frames), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at that index must be of type CONSTANT_VirtualMethodref (6.8.2 CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref), 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. Finally, if the
resolved method is protected, and it is a member of a superclass of the current class, and the method is
not declared in the same package as the current class, then 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 (2.2.1.4 Unsupported Classes) 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 Contexts) is not the context (3.4 Contexts) of the object referenced by objectref. The exact
circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment
Specification, Java Card Platform, v3.1, Classic Edition*. If the current context is not the object’s context
and the Java Card RE permits invocation of the method, the invokevirtual instruction will cause a context
switch (3.4 Contexts) 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.

### 7.5.57.1 Virtual Method Resolution

The resolution of the virtual method is defined by the following algorithm. The virtual machine is not
required to exactly perform the following steps provided the result is the same.

1. Verify access permission from current context to object O on the stack
2. Extract the class C1 and token T1 from the constant pool entry
   CONSTANT_VirtualMethodref
3. Get the dynamic type C of the object O and lookup the method in the hierarchy of the class, i.e.:
a. Determine a token $T$ in class $C$ that denotes the same method as token $T_1$ in class $C_1$

b. If the token $T$ does not exist or if $T$ is less than
   
   $C$.public_virtual_method_table_base or if
   
   $C$.public_virtual_method_table[$T -
   
   C$.public_virtual_method_table_base] is 0xFFFF, then recursively
   
   continue the lookup at step a, with $C$ being now set to the super-class of $C$

c. Otherwise,
   
   $C$.public_virtual_method_table[$T -
   
   C$.public_virtual_method_table_base] is the location in class $C$ of the method $M$
   
   to be invoked

4. If no method implementation has been found in the hierarchy or if a method implementation
   has been found but has the ACC_ABSTRACT flag, then the virtual machine must exit the virtual
   method resolution and handle this as an error. See section 2.3.3.1.

7.5.58 ior

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.

7.5.59 irem

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 \(\text{value1} - (\text{value1} / \text{value2}) \times \text{value2}\). The result is pushed onto the operand stack.

The result of the irem instruction is such that \((a/b) \times b + (a\%b)\) 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.

7.5.60 ireturn

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 Frames) 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.

7.5.61 ishl
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.

7.5.62 ishr
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 \( \lfloor \frac{\text{value1}}{2^s} \rfloor \), 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.

If a virtual machine does not support the int data type, the ishr instruction will not be available.

7.5.63 istore
Store int into local variable

Format

\texttt{istore} \\
\texttt{index}

Forms

\texttt{istore} = 42 (0x2a)

Stack

\ldots, \texttt{value.word1}, \texttt{value.word2} \rightarrow \ldots

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 Frames). 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.

7.5.64 istore_<n>
Store int into local variable

Format

\texttt{istore_<n>}

Forms
istore_0 = 51 (0x33)
istore_1 = 52 (0x34)
istore_2 = 53 (0x35)
istore_3 = 54 (0x36)

**Stack**

..., value.word1, value.word2 -> ...

**Description**

Both <i>n</i> and <i>n</i> + 1 must be a valid indices into the local variables of the current frame (3.5 Frames). 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_<i>n</i> instruction will not be available.

### 7.5.65 isub

Subtract int

**Format**

<code>isub</code>

**Forms**

<code>isub</code>

**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, <i>a</i> − <i>b</i> produces the same result as <i>a</i> + (−<i>b</i>). 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.

7.5.66 itableswitch
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.

7.5.67 iushr
Logical shift right int

Format

\[ \text{iushr} \]

Forms

\[ \text{iushr} = 82 \ (0x52) \]

Stack

\[ \ldots, \text{value1.word1, value1.word2, value2.word1, value2.word2} \rightarrow \ldots, \text{result.word1, result.word2} \]

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.

7.5.68 ixor
Boolean XOR int

Format

\[ \text{ixor} \]

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.

7.5.69 jsr
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.
7.5.70 **new**  
Create new object

**Format**

```
new  
indexbyte1  
indexbyte2
```

**Forms**

```
new = 143 (0xf)
```

**Stack**

```
..., objectref
```

**Description**

The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the constant pool of the current package (3.5 Frames), 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` (6.8.1 `CONSTANT_Classref`), 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.

7.5.71 **newarray**  
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:

**Table 7-4: Array Values**

<table>
<thead>
<tr>
<th>Array Type</th>
<th>atype</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_BOOLEAN</td>
<td>10</td>
</tr>
<tr>
<td>T_BYTE</td>
<td>11</td>
</tr>
<tr>
<td>T_SHORT</td>
<td>12</td>
</tr>
<tr>
<td>T_INT</td>
<td>13</td>
</tr>
</tbody>
</table>

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).

**7.5.72 nop**

Do nothing

**Format**

*nop*

**Forms**

*nop = 0 (0x0)*

**Stack**

No change

**Description**
Do nothing.

7.5.73 pop
Pop top operand stack word

Format

pop

Forms

pop = 59 (0x3b)

Stack

..., word -> ...

Description

The top word is popped from the operand stack. The pop instruction must not be used unless the word contains a 16-bit data type.

Notes

The pop instruction operates on an untyped word, ignoring the type of data it contains.

7.5.74 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.

### 7.5.75 `putfield_<t>`
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 Frames). The constant pool item at the index must be of type `CONSTANT_InstanceFieldref` (6.8.2 `CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref`), a reference to a class and a field token.

The class of `objectref` must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of `objectref` must be either the current class or a subclass of the current class. If the field is final, it must be declared in 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

value must be of a type that is assignment compatible with the field descriptor (`t`) type.
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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the putfield_i instruction will not be available.

### 7.5.76 putfield_<t>_this

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**

---

22 The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.
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 Frames). The constant pool item at the index must be of type CONSTANT_InstanceFieldref (6.8.2 CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class. If the field is final, it must be declared in 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

value must be of a type that is assignment compatible with the field descriptor (t) type.

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\(^{23}\). 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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the putfield_i_this instruction will not be available.

---

\(^{23}\) The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance.
7.5.77 putfield_<t>_w
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 (3.4 Contexts), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at the index must be of type CONSTANT_InstanceFieldref (6.8.2 CONSTANT_InstanceFieldref, CONSTANT_VirtualMethodref, CONSTANT_SuperMethodref), a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class. If the field is final, it must be declared in 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

value must be of a type that is assignment compatible with the field descriptor (t) type.
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\(^{24}\). 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 Contexts) is not the owning context (3.4 Contexts) of the object referenced by objectref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

**7.5.78 putstatic\(_<t>_\)**

Set static field in class

**Format**

\[ \text{putstatic\(_<t>_\)}, \text{indexbyte1}, \text{indexbyte2} \]

**Forms**

- putstatic\(_a = 127 (0x7f)\)
- putstatic\(_b = 128 (0x80)\)
- putstatic\(_s = 129 (0x81)\)
- putstatic\(_i = 130 (0x82)\)

**Stack**

\[ ..., \text{value} \rightarrow ... \]

**OR**

\[^{24}\text{The offset may be computed by adding the field token value to the size of an instance of the immediate superclass. However, this method is not required by this specification. A Java Card virtual machine may define any mapping from token value to offset into an instance. If a virtual machine does not support the int data type, the putfield\(_i_w\) instruction will not be available.}\]
..., 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 Frames), where the value of the index is (indexbyte1 << 8) | indexbyte2. The constant pool item at the index must be of type CONSTANT_StaticFieldref (6.8.3 CONSTANT_StaticFieldref and CONSTANT_StaticMethodref), a reference to a static field. If the field is final, it must be declared in 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

value must be of a type that is assignment compatible with the field descriptor (t) type.

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 Contexts) is not the owning context (3.4 Contexts) of the object being stored in the field. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

If a virtual machine does not support the int data type, the putstatic_i instruction will not be available.

**7.5.79 ret**

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 Frames). 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.

7.5.80 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.

7.5.81 s2b
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.

7.5.82 s2i
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.

7.5.83 sadd
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.

7.5.84 saload
Load short from array

Format

saload

Forms

saload = 38 (0x26)

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.

Otherwise if the arrayref references a write-only array view, the saload instruction throws a SecurityException.

Notes

In some circumstances, the saload instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact
circumstances when the exception will be thrown are specified in Chapter 6 of the *Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition*.

7.5.85 *s*and

Boolean AND short

**Format**

\( s \text{and} \)

**Forms**

\( s \text{and} = 83 \text{ (0x53)} \)

**Stack**

\( ..., \text{value1, value2 -> ..., result} \)

**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.

7.5.86 *s*astore

Store into short array

**Format**

\( \text{sastore} \)

**Forms**

\( \text{sastore} = 57 \text{ (0x39)} \)

**Stack**

\( ..., \text{arrayref, index, value -> ...} \)

**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.

If the array referenced by arrayref is integrity-sensitive, its integrity is checked before the value is stored. The integrity control element is updated when the value is stored. The whole operation (value storage and the integrity control element update) is performed atomically.

**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.

Otherwise if the array referenced by arrayref is integrity-sensitive and an inconsistency is detected during the array integrity check, the sastore instruction throws a SecurityException.

Otherwise if the arrayref references a read-only array view, the sastore instruction throws a SecurityException.

Notes

In some circumstances, the sastore instruction may throw a SecurityException if the current context (3.4 Contexts) is not the owning context (3.4 Contexts) of the array referenced by arrayref. The exact circumstances when the exception will be thrown are specified in Chapter 6 of the Runtime Environment Specification, Java Card Platform, v3.1, Classic Edition.

7.5.87 sconst_<s>
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.

7.5.88 sdiv
Divide short

Format
\textit{sdiv}

**Forms**

\texttt{sdiv = 71 (0x47)}

**Stack**

\texttt{..., 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 \texttt{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 \texttt{n/d} is a short value \texttt{q} whose magnitude is as large as possible while satisfying \texttt{| d \cdot q | \leq | n |}. Moreover, \texttt{q} is a positive when \texttt{| n | \geq | d |} and \texttt{n} and \texttt{d} have the same sign, but \texttt{q} is negative when \texttt{| n | \geq | d |} and \texttt{n} and \texttt{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 \texttt{-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, \textit{sdiv} throws an \texttt{ArithmeticException}.

\textbf{7.5.89 sinc}

Increment local short variable by constant

**Format**

\texttt{sinc index const}

**Forms**

\texttt{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 Frames). 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.

**7.5.90 sinc_w**
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 Frames). 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.

**7.5.91 sipush**
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.

### 7.5.92 sload

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 Frames). The local variable at index must contain a short. The value in the local variable at index is pushed onto the operand stack.

### 7.5.93 sload_<n>

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 Frames). 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.

7.5.94 slookupswitch
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.

### 7.5.95 smul

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.

### 7.5.96 sneg

Negate short

**Format**

```\nsneg
```

**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.

7.5.97 sor
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.

7.5.98 srem
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 srem 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.

7.5.99 sreturn
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 Frames) 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.

7.5.100 sshl
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.

The mask value of 0x1f allows shifting beyond the range of a 16-bit short value. It is used by this instruction, however, to ensure results equal to those generated by the Java instruction ishl.

7.5.101 sshr
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 \rceil, 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.

The mask value of 0x1f allows shifting beyond the range of a 16-bit short value. It is used by this instruction, however, to ensure results equal to those generated by the Java instruction ishr.

### 7.5.102 sspush
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.

### 7.5.103 sstore
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 Frames). 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.

7.5.104 sstore_<n>
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 Frames). 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.

7.5.105 ssub
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.

### 7.5.106 stableswitch

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.

7.5.107 sushr
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.

The mask value of 0x1f allows shifting beyond the range of a 16-bit short value. It is used by this instruction, however, to ensure results equal to those generated by the Java instruction iushr.

7.5.108 swap_x
Swap top two operand stack words

---

25 Sign extension to 32 bits ensures that the result computed by this instruction will be exactly equal to that computed by the Java iushr instruction, regardless of the input values. In a JavaCard virtual machine the expression “0xffff >>> 0x01” yields 0xffff, where “>>>” is performed by the sushr instruction. The same result is rendered by a Java virtual machine.
**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 `m+n` each contain either a 16-bit data type or a 32-bit data type.

**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.

**7.5.109 sxor**

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.
# Tables of Instructions

The following pages contain lists of the virtual machine instructions recognized by the Java Card platform, organized by opcode value (Table 8-1) and by opcode mnemonic (Table 8-2).

## 8.1 Instructions by Opcode Value

The following table shows the opcode values for the instructions recognized by the Java Card Platform.

### Table 8-1: Instructions by Opcode Value

<table>
<thead>
<tr>
<th>dec</th>
<th>hex</th>
<th>mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>nop</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>aconst_null</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>sconst_m1</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>sconst_0</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>sconst_1</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>sconst_2</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>sconst_3</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>sconst_4</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>sconst_5</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>icont_m1</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>icont_0</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>icont_1</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>icont_2</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>icont_3</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>icont_4</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>icont_5</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>bspush</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>sspush</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>bipush</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>sipush</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>iipush</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>aload</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>sload</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>iload</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>aload_0</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>aload_1</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>aload_2</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>aload_3</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>sload_0</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>sload_1</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>sload_2</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>sload_3</td>
</tr>
<tr>
<td>dec</td>
<td>hex</td>
<td>mnemonic</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>iload_0</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>iload_1</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>iload_2</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>iload_3</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>aaload</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>baload</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>saload</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>iaload</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>astore</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>sstore</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>istore</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>astore_0</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>astore_1</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>astore_2</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>astore_3</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>sstore_0</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>sstore_1</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>sstore_2</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>sstore_3</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>istore_0</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>istore_1</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>istore_2</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>istore_3</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>aastore</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>bastore</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>sastore</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>iastore</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>pop</td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>pop2</td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>dup</td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>dup2</td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>dup_x</td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>swap_x</td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>sadd</td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>iadd</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>ssub</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>isub</td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>smul</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>imul</td>
</tr>
<tr>
<td>71</td>
<td>47</td>
<td>sdiv</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>idiv</td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td>srem</td>
</tr>
<tr>
<td>74</td>
<td>4A</td>
<td>irem</td>
</tr>
<tr>
<td>75</td>
<td>4B</td>
<td>sneg</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>ineg</td>
</tr>
<tr>
<td>dec</td>
<td>hex</td>
<td>mnemonic</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>----------</td>
</tr>
<tr>
<td>77</td>
<td>4D</td>
<td>sshl</td>
</tr>
<tr>
<td>78</td>
<td>4E</td>
<td>ishl</td>
</tr>
<tr>
<td>79</td>
<td>4F</td>
<td>sshr</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>ishr</td>
</tr>
<tr>
<td>81</td>
<td>51</td>
<td>sushr</td>
</tr>
<tr>
<td>82</td>
<td>52</td>
<td>iuushr</td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td>sand</td>
</tr>
<tr>
<td>84</td>
<td>54</td>
<td>iand</td>
</tr>
<tr>
<td>85</td>
<td>55</td>
<td>sor</td>
</tr>
<tr>
<td>86</td>
<td>56</td>
<td>ior</td>
</tr>
<tr>
<td>87</td>
<td>57</td>
<td>sxor</td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td>ixor</td>
</tr>
<tr>
<td>89</td>
<td>59</td>
<td>sinc</td>
</tr>
<tr>
<td>90</td>
<td>5A</td>
<td>iinc</td>
</tr>
<tr>
<td>91</td>
<td>5B</td>
<td>s2b</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>s2i</td>
</tr>
<tr>
<td>93</td>
<td>5D</td>
<td>i2b</td>
</tr>
<tr>
<td>94</td>
<td>5E</td>
<td>i2s</td>
</tr>
<tr>
<td>95</td>
<td>5F</td>
<td>icmp</td>
</tr>
<tr>
<td>96</td>
<td>60</td>
<td>ifeq</td>
</tr>
<tr>
<td>97</td>
<td>61</td>
<td>ifne</td>
</tr>
<tr>
<td>98</td>
<td>62</td>
<td>iflt</td>
</tr>
<tr>
<td>99</td>
<td>63</td>
<td>ifge</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>ifgt</td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>ifle</td>
</tr>
<tr>
<td>102</td>
<td>66</td>
<td>ifnull</td>
</tr>
<tr>
<td>103</td>
<td>67</td>
<td>ifnonnull</td>
</tr>
<tr>
<td>104</td>
<td>68</td>
<td>if_acmpeq</td>
</tr>
<tr>
<td>105</td>
<td>69</td>
<td>if_acmpne</td>
</tr>
<tr>
<td>106</td>
<td>6A</td>
<td>if scmpeq</td>
</tr>
<tr>
<td>107</td>
<td>6B</td>
<td>if scmpne</td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>if scmplt</td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>if scmpge</td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>if scmpgt</td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>if scmple</td>
</tr>
<tr>
<td>112</td>
<td>70</td>
<td>goto</td>
</tr>
<tr>
<td>113</td>
<td>71</td>
<td>jsr</td>
</tr>
<tr>
<td>114</td>
<td>72</td>
<td>ret</td>
</tr>
<tr>
<td>115</td>
<td>73</td>
<td>stableswitch</td>
</tr>
<tr>
<td>116</td>
<td>74</td>
<td>itableswitch</td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td>slookupswitch</td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td>ilookupswitch</td>
</tr>
<tr>
<td>119</td>
<td>77</td>
<td>areturn</td>
</tr>
<tr>
<td>120</td>
<td>78</td>
<td>sreturn</td>
</tr>
<tr>
<td>121</td>
<td>79</td>
<td>ireturn</td>
</tr>
<tr>
<td>dec</td>
<td>hex</td>
<td>mnemonic</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-------------------</td>
</tr>
<tr>
<td>122</td>
<td>7A</td>
<td>return</td>
</tr>
<tr>
<td>123</td>
<td>7B</td>
<td>getstatic_a</td>
</tr>
<tr>
<td>124</td>
<td>7C</td>
<td>getstatic_b</td>
</tr>
<tr>
<td>125</td>
<td>7D</td>
<td>getstatic_s</td>
</tr>
<tr>
<td>126</td>
<td>7E</td>
<td>getstatic_i</td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
<td>putstatic_a</td>
</tr>
<tr>
<td>128</td>
<td>80</td>
<td>putstatic_b</td>
</tr>
<tr>
<td>129</td>
<td>81</td>
<td>putstatic_s</td>
</tr>
<tr>
<td>130</td>
<td>82</td>
<td>putstatic_i</td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td>getfield_a</td>
</tr>
<tr>
<td>132</td>
<td>84</td>
<td>getfield_b</td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td>getfield_s</td>
</tr>
<tr>
<td>134</td>
<td>86</td>
<td>getfield_i</td>
</tr>
<tr>
<td>135</td>
<td>87</td>
<td>putfield_a</td>
</tr>
<tr>
<td>136</td>
<td>88</td>
<td>putfield_b</td>
</tr>
<tr>
<td>137</td>
<td>89</td>
<td>putfield_s</td>
</tr>
<tr>
<td>138</td>
<td>8A</td>
<td>putfield_i</td>
</tr>
<tr>
<td>139</td>
<td>8B</td>
<td>invokevirtual</td>
</tr>
<tr>
<td>140</td>
<td>8C</td>
<td>invokespecial</td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td>invokestatic</td>
</tr>
<tr>
<td>142</td>
<td>8E</td>
<td>invokeinterface</td>
</tr>
<tr>
<td>143</td>
<td>8F</td>
<td>new</td>
</tr>
<tr>
<td>144</td>
<td>90</td>
<td>newarray</td>
</tr>
<tr>
<td>145</td>
<td>91</td>
<td>anewarray</td>
</tr>
<tr>
<td>146</td>
<td>92</td>
<td>arraylength</td>
</tr>
<tr>
<td>147</td>
<td>93</td>
<td>athrow</td>
</tr>
<tr>
<td>148</td>
<td>94</td>
<td>checkcast</td>
</tr>
<tr>
<td>149</td>
<td>95</td>
<td>instanceof</td>
</tr>
<tr>
<td>150</td>
<td>96</td>
<td>instanceof_w</td>
</tr>
<tr>
<td>151</td>
<td>97</td>
<td>iinc_w</td>
</tr>
<tr>
<td>152</td>
<td>98</td>
<td>ifeq_w</td>
</tr>
<tr>
<td>153</td>
<td>99</td>
<td>ifne_w</td>
</tr>
<tr>
<td>154</td>
<td>9A</td>
<td>iflt_w</td>
</tr>
<tr>
<td>155</td>
<td>9B</td>
<td>ifge_w</td>
</tr>
<tr>
<td>156</td>
<td>9C</td>
<td>ifgt_w</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>ifle_w</td>
</tr>
<tr>
<td>158</td>
<td>9E</td>
<td>ifnonnull_w</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>ifnonnull_w</td>
</tr>
<tr>
<td>160</td>
<td>A0</td>
<td>if_acmpeq_w</td>
</tr>
<tr>
<td>161</td>
<td>A1</td>
<td>if_acmpne_w</td>
</tr>
<tr>
<td>162</td>
<td>A2</td>
<td>if_scmpeq_w</td>
</tr>
<tr>
<td>163</td>
<td>A3</td>
<td>if_scmpne_w</td>
</tr>
<tr>
<td>164</td>
<td>A4</td>
<td>if_scmplt_w</td>
</tr>
<tr>
<td>165</td>
<td>A5</td>
<td>if_scmpge_w</td>
</tr>
<tr>
<td>166</td>
<td>A6</td>
<td>if__scmpgt_w</td>
</tr>
</tbody>
</table>
### 8.2 Instructions by Opcode Mnemonic

The following table shows the opcode mnemonics for the instructions recognized by the Java Card Platform.

**Table 8-2: Instructions by Opcode Mnemonic**

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>dec</th>
<th>hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>aalborg</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td>aastore</td>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>aconst_null</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>aload</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>aload_0</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td>aload_1</td>
<td>25</td>
<td>19</td>
</tr>
<tr>
<td>aload_2</td>
<td>26</td>
<td>1A</td>
</tr>
<tr>
<td>aload_3</td>
<td>27</td>
<td>1B</td>
</tr>
<tr>
<td>anewarray</td>
<td>145</td>
<td>91</td>
</tr>
<tr>
<td>areturn</td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>arraylength</td>
<td>146</td>
<td>92</td>
</tr>
<tr>
<td>astore</td>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>astore_0</td>
<td>43</td>
<td>2B</td>
</tr>
<tr>
<td>astore_1</td>
<td>44</td>
<td>2C</td>
</tr>
<tr>
<td>astore_2</td>
<td>45</td>
<td>2D</td>
</tr>
<tr>
<td>astore_3</td>
<td>46</td>
<td>2E</td>
</tr>
<tr>
<td>mnemonic</td>
<td>dec</td>
<td>hex</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>athrow</td>
<td>147</td>
<td>93</td>
</tr>
<tr>
<td>baload</td>
<td>37</td>
<td>25</td>
</tr>
<tr>
<td>bastore</td>
<td>56</td>
<td>38</td>
</tr>
<tr>
<td>bipush</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>bspush</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>checkcast</td>
<td>148</td>
<td>94</td>
</tr>
<tr>
<td>dup</td>
<td>61</td>
<td>3D</td>
</tr>
<tr>
<td>dup_x</td>
<td>63</td>
<td>3F</td>
</tr>
<tr>
<td>dup2</td>
<td>62</td>
<td>3E</td>
</tr>
<tr>
<td>getfield_a</td>
<td>131</td>
<td>83</td>
</tr>
<tr>
<td>getfield_a_this</td>
<td>173</td>
<td>AD</td>
</tr>
<tr>
<td>getfield_a_w</td>
<td>169</td>
<td>A9</td>
</tr>
<tr>
<td>getfield_b</td>
<td>132</td>
<td>84</td>
</tr>
<tr>
<td>getfield_b_this</td>
<td>174</td>
<td>AE</td>
</tr>
<tr>
<td>getfield_b_w</td>
<td>170</td>
<td>AA</td>
</tr>
<tr>
<td>getfield_i</td>
<td>134</td>
<td>86</td>
</tr>
<tr>
<td>getfield_i_this</td>
<td>176</td>
<td>B0</td>
</tr>
<tr>
<td>getfield_i_w</td>
<td>172</td>
<td>AC</td>
</tr>
<tr>
<td>getfield_s</td>
<td>133</td>
<td>85</td>
</tr>
<tr>
<td>getfield_s_this</td>
<td>175</td>
<td>AF</td>
</tr>
<tr>
<td>getfield_s_w</td>
<td>171</td>
<td>AB</td>
</tr>
<tr>
<td>getstatic_a</td>
<td>123</td>
<td>7B</td>
</tr>
<tr>
<td>getstatic_b</td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>getstatic_i</td>
<td>126</td>
<td>7E</td>
</tr>
<tr>
<td>getstatic_s</td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>goto</td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>goto_w</td>
<td>168</td>
<td>A8</td>
</tr>
<tr>
<td>i2b</td>
<td>93</td>
<td>5D</td>
</tr>
<tr>
<td>i2s</td>
<td>94</td>
<td>5E</td>
</tr>
<tr>
<td>iadd</td>
<td>66</td>
<td>42</td>
</tr>
<tr>
<td>iaload</td>
<td>39</td>
<td>27</td>
</tr>
<tr>
<td>iand</td>
<td>84</td>
<td>54</td>
</tr>
<tr>
<td>iastore</td>
<td>58</td>
<td>3A</td>
</tr>
<tr>
<td>icmp</td>
<td>95</td>
<td>5F</td>
</tr>
<tr>
<td>iconst_0</td>
<td>10</td>
<td>0A</td>
</tr>
<tr>
<td>iconst_1</td>
<td>11</td>
<td>0B</td>
</tr>
<tr>
<td>iconst_2</td>
<td>12</td>
<td>0C</td>
</tr>
<tr>
<td>iconst_3</td>
<td>13</td>
<td>0D</td>
</tr>
<tr>
<td>iconst_4</td>
<td>14</td>
<td>0E</td>
</tr>
<tr>
<td>iconst_5</td>
<td>15</td>
<td>0F</td>
</tr>
<tr>
<td>iconst_m1</td>
<td>9</td>
<td>09</td>
</tr>
<tr>
<td>idiv</td>
<td>72</td>
<td>48</td>
</tr>
<tr>
<td>if_acmpeq</td>
<td>104</td>
<td>68</td>
</tr>
<tr>
<td>if_acmpne_w</td>
<td>160</td>
<td>A0</td>
</tr>
<tr>
<td>if_acmpne</td>
<td>105</td>
<td>69</td>
</tr>
<tr>
<td>mnemonic</td>
<td>dec</td>
<td>hex</td>
</tr>
<tr>
<td>------------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>if_acmpne_w</td>
<td>161</td>
<td>A1</td>
</tr>
<tr>
<td>if_scmpeq</td>
<td>106</td>
<td>6A</td>
</tr>
<tr>
<td>if_scmpeq_w</td>
<td>162</td>
<td>A2</td>
</tr>
<tr>
<td>if_scmpge</td>
<td>109</td>
<td>6D</td>
</tr>
<tr>
<td>if_scmpge_w</td>
<td>165</td>
<td>A5</td>
</tr>
<tr>
<td>if_scmpeq</td>
<td>110</td>
<td>6E</td>
</tr>
<tr>
<td>if_scmpeq_w</td>
<td>166</td>
<td>A6</td>
</tr>
<tr>
<td>if_scmpne</td>
<td>111</td>
<td>6F</td>
</tr>
<tr>
<td>if_scmpne_w</td>
<td>167</td>
<td>A7</td>
</tr>
<tr>
<td>if_scmpge</td>
<td>108</td>
<td>6C</td>
</tr>
<tr>
<td>if_scmpge_w</td>
<td>164</td>
<td>A4</td>
</tr>
<tr>
<td>if_scmpne</td>
<td>107</td>
<td>6B</td>
</tr>
<tr>
<td>if_scmpne_w</td>
<td>163</td>
<td>A3</td>
</tr>
<tr>
<td>ifeq</td>
<td>96</td>
<td>60</td>
</tr>
<tr>
<td>ifeq_w</td>
<td>152</td>
<td>98</td>
</tr>
<tr>
<td>ifge</td>
<td>99</td>
<td>63</td>
</tr>
<tr>
<td>ifge_w</td>
<td>155</td>
<td>9B</td>
</tr>
<tr>
<td>ifgt</td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>ifgt_w</td>
<td>156</td>
<td>9C</td>
</tr>
<tr>
<td>ifle</td>
<td>101</td>
<td>65</td>
</tr>
<tr>
<td>ifle_w</td>
<td>157</td>
<td>9D</td>
</tr>
<tr>
<td>iflt</td>
<td>98</td>
<td>62</td>
</tr>
<tr>
<td>iflt_w</td>
<td>154</td>
<td>9A</td>
</tr>
<tr>
<td>ifne</td>
<td>97</td>
<td>61</td>
</tr>
<tr>
<td>ifne_w</td>
<td>153</td>
<td>99</td>
</tr>
<tr>
<td>ifnonnull</td>
<td>103</td>
<td>67</td>
</tr>
<tr>
<td>ifnonnull_w</td>
<td>159</td>
<td>9F</td>
</tr>
<tr>
<td>ifnull</td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>ifnull_w</td>
<td>158</td>
<td>9E</td>
</tr>
<tr>
<td>iinc</td>
<td>90</td>
<td>5A</td>
</tr>
<tr>
<td>iinc_w</td>
<td>151</td>
<td>97</td>
</tr>
<tr>
<td>iipush</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>iload</td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>iload_0</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>iload_1</td>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>iload_2</td>
<td>34</td>
<td>22</td>
</tr>
<tr>
<td>iload_3</td>
<td>35</td>
<td>23</td>
</tr>
<tr>
<td>ilookupswitch</td>
<td>118</td>
<td>76</td>
</tr>
<tr>
<td>imul</td>
<td>70</td>
<td>46</td>
</tr>
<tr>
<td>ineg</td>
<td>76</td>
<td>4C</td>
</tr>
<tr>
<td>instanceof</td>
<td>149</td>
<td>95</td>
</tr>
<tr>
<td>invokeinterface</td>
<td>142</td>
<td>8E</td>
</tr>
<tr>
<td>invokespecial</td>
<td>140</td>
<td>8C</td>
</tr>
<tr>
<td>invokestatic</td>
<td>141</td>
<td>8D</td>
</tr>
<tr>
<td>invokevirtual</td>
<td>139</td>
<td>8B</td>
</tr>
<tr>
<td>mnemonic</td>
<td>dec</td>
<td>hex</td>
</tr>
<tr>
<td>------------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>ior</td>
<td>86</td>
<td>56</td>
</tr>
<tr>
<td>irem</td>
<td>74</td>
<td>4A</td>
</tr>
<tr>
<td>ireturn</td>
<td>121</td>
<td>79</td>
</tr>
<tr>
<td>ishl</td>
<td>78</td>
<td>4E</td>
</tr>
<tr>
<td>ishr</td>
<td>80</td>
<td>50</td>
</tr>
<tr>
<td>istore</td>
<td>42</td>
<td>2A</td>
</tr>
<tr>
<td>istore_0</td>
<td>51</td>
<td>33</td>
</tr>
<tr>
<td>istore_1</td>
<td>52</td>
<td>34</td>
</tr>
<tr>
<td>istore_2</td>
<td>53</td>
<td>35</td>
</tr>
<tr>
<td>istore_3</td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>isub</td>
<td>68</td>
<td>44</td>
</tr>
<tr>
<td>itableswitch</td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>iushr</td>
<td>82</td>
<td>52</td>
</tr>
<tr>
<td>ixor</td>
<td>88</td>
<td>58</td>
</tr>
<tr>
<td>jsr</td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>new</td>
<td>143</td>
<td>8F</td>
</tr>
<tr>
<td>newarray</td>
<td>144</td>
<td>90</td>
</tr>
<tr>
<td>nop</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>pop</td>
<td>59</td>
<td>3B</td>
</tr>
<tr>
<td>pop2</td>
<td>60</td>
<td>3C</td>
</tr>
<tr>
<td>putfield_a</td>
<td>135</td>
<td>87</td>
</tr>
<tr>
<td>putfield_a_this</td>
<td>177</td>
<td>B1</td>
</tr>
<tr>
<td>putfield_a_w</td>
<td>136</td>
<td>88</td>
</tr>
<tr>
<td>putfield_b</td>
<td>182</td>
<td>B6</td>
</tr>
<tr>
<td>putfield_b_this</td>
<td>178</td>
<td>B2</td>
</tr>
<tr>
<td>putfield_i</td>
<td>138</td>
<td>8A</td>
</tr>
<tr>
<td>putfield_i_this</td>
<td>184</td>
<td>B8</td>
</tr>
<tr>
<td>putfield_i_w</td>
<td>180</td>
<td>B4</td>
</tr>
<tr>
<td>putfield_s</td>
<td>137</td>
<td>89</td>
</tr>
<tr>
<td>putfield_s_this</td>
<td>183</td>
<td>B7</td>
</tr>
<tr>
<td>putstatic_a</td>
<td>127</td>
<td>7F</td>
</tr>
<tr>
<td>putstatic_b</td>
<td>128</td>
<td>80</td>
</tr>
<tr>
<td>putstatic_i</td>
<td>130</td>
<td>82</td>
</tr>
<tr>
<td>putstatic_s</td>
<td>129</td>
<td>81</td>
</tr>
<tr>
<td>ret</td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>return</td>
<td>122</td>
<td>7A</td>
</tr>
<tr>
<td>s2b</td>
<td>91</td>
<td>5B</td>
</tr>
<tr>
<td>s2i</td>
<td>92</td>
<td>5C</td>
</tr>
<tr>
<td>sadd</td>
<td>65</td>
<td>41</td>
</tr>
<tr>
<td>saload</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>sand</td>
<td>83</td>
<td>53</td>
</tr>
<tr>
<td>sastore</td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>sconst_0</td>
<td>3</td>
<td>03</td>
</tr>
<tr>
<td>mnemonic</td>
<td>dec</td>
<td>hex</td>
</tr>
<tr>
<td>---------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>sconst_1</td>
<td>4</td>
<td>04</td>
</tr>
<tr>
<td>sconst_2</td>
<td>5</td>
<td>05</td>
</tr>
<tr>
<td>sconst_3</td>
<td>6</td>
<td>06</td>
</tr>
<tr>
<td>sconst_4</td>
<td>7</td>
<td>07</td>
</tr>
<tr>
<td>sconst_5</td>
<td>8</td>
<td>08</td>
</tr>
<tr>
<td>sconst_m1</td>
<td>2</td>
<td>02</td>
</tr>
<tr>
<td>sdiv</td>
<td>71</td>
<td>47</td>
</tr>
<tr>
<td>sinc</td>
<td>89</td>
<td>59</td>
</tr>
<tr>
<td>sinc_w</td>
<td>150</td>
<td>96</td>
</tr>
<tr>
<td>sipush</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>sload</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>sload_0</td>
<td>28</td>
<td>1C</td>
</tr>
<tr>
<td>sload_1</td>
<td>29</td>
<td>1D</td>
</tr>
<tr>
<td>sload_2</td>
<td>30</td>
<td>1E</td>
</tr>
<tr>
<td>sload_3</td>
<td>31</td>
<td>1F</td>
</tr>
<tr>
<td>slookupswitch</td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>smul</td>
<td>69</td>
<td>45</td>
</tr>
<tr>
<td>sneg</td>
<td>75</td>
<td>4B</td>
</tr>
<tr>
<td>sor</td>
<td>85</td>
<td>55</td>
</tr>
<tr>
<td>srem</td>
<td>73</td>
<td>49</td>
</tr>
<tr>
<td>sreturn</td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>sshl</td>
<td>77</td>
<td>4D</td>
</tr>
<tr>
<td>sshr</td>
<td>79</td>
<td>4F</td>
</tr>
<tr>
<td>sspush</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>sstore</td>
<td>41</td>
<td>29</td>
</tr>
<tr>
<td>sstore_0</td>
<td>47</td>
<td>2F</td>
</tr>
<tr>
<td>sstore_1</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>sstore_2</td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>sstore_3</td>
<td>50</td>
<td>32</td>
</tr>
<tr>
<td>ssub</td>
<td>67</td>
<td>43</td>
</tr>
<tr>
<td>stableswitch</td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>sushr</td>
<td>81</td>
<td>51</td>
</tr>
<tr>
<td>swap_x</td>
<td>64</td>
<td>40</td>
</tr>
<tr>
<td>sxor</td>
<td>87</td>
<td>57</td>
</tr>
</tbody>
</table>
Glossary

A

active applet instance
an applet instance that is selected on at least one of the logical channels.

AID (application identifier)
defined by ISO 7816, a string used to uniquely identify card applications and certain types of files in card file systems. An AID consists of two distinct pieces: a 5-byte RID (resource identifier) and a 0 to 11-byte PIX (proprietary identifier extension). The RID is a resource identifier assigned to companies by ISO. The PIX identifiers are assigned by companies.

A unique AID is assigned to each CAP file and public packages in a CAP file. In addition, a unique AID is assigned to each applet in the CAP file. The AID for the CAP file, the package AID of every public package in a CAP file and the default AID for each applet defined in the CAP file are specified in the CAP file. They are supplied to the converter when the CAP file is generated.

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

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

applet
within the context of this document, a Java Card applet, which is the basic unit of selection, context, functionality, and security in Java Card technology.

applet application
an application that consists of one or more applets.

applet framework
an API that enables applet applications to be built.

applet developer
a person creating an applet using Java Card technology.
**applet execution context**
currently active applet owner identifier.

**applet firewall**
the mechanism that prevents unauthorized accesses to objects in contexts other than currently active context.

**applet CAP file**
a CAP file that contains one or more applet packages. See applet package.

**applet package**
a Java programming language package that contains one or more non-abstract classes that extend the javacard.framework.Applet class. See also library package.

**assigned logical channel**
the logical channel on which the applet instance is either the active applet instance or will become the active applet instance.

**atomic operation**
an operation that either completes in its entirety or no part of the operation completes at all.

**atomicity**
state in which a particular operation is atomic. Atomicity of data updates guarantee that data are not corrupted in case of power loss or card removal.

**ATR**
an acronym for Answer to Reset. An ATR is a string of bytes sent by the Java Card platform after a reset condition.

**authentication**
the process of establishing or confirming an application or a user as authentic using some sort of credentials
B

**basic logical channel**

logical channel 0, the only channel that is active at card reset in the APDU application environment. This channel is permanent and can never be closed.

**big-endian**

a technique of storing multibyte data where the high-order bytes come first. For example, given an 8-bit data item stored in big-endian order, the first bit read is considered the high bit.

**binary compatibility**

in a Java Card system, a change in a Java programming language package in a Java Card CAP file 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 files of packages included in the preexisting CAP file can link with the new CAP file without errors.

**bytecode**

machine-independent code generated by the compiler and executed by the Java virtual machine.

C

**CAD**

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

**CAP file**

Standard file format containing a binary representation of a shared library (library CAP file) or an application with its libraries that might be exported or not (applet CAP file).

A CAP file represents a module, which is a unit of code, made of one or more Java packages, with dependencies and list of exported packages and an assigned name (AID) for lifecycle management. Its structure is made of multiple CAP components deployed within a JAR file.

When a CAP file containing application(s) is deployed on a Java Card platform, it is assigned a new unique group context that must be associated with any application instance created from code within this application module.

**CAP file component**

A Java Card platform CAP file consists of a set of components, which represent a set of one or more Java programming language packages. Each component describes a set of elements or an aspect of the CAP.
A complete CAP file must contain all of the required components: Header, Directory, Import, Constant Pool, Method, Static Field, and Reference Location.

The following components are conditionally included or optional: the Applet, Export, Static Resources and Debug. The Applet component is included only if one or more applets are defined in one or more packages in the CAP file. The Export component is included only if one or more packages are public and exported allowing classes in other packages to import elements from them. The Static Resources component is included only if static resources are embedded in the CAP file. The Debug component is optional. It contains all of the data necessary for debugging.

card session

A card session begins with the insertion of the card into the CAD. The card is then able to exchange streams of APDUs with the CAD. The card session ends when the card is removed from the CAD.

cast

The explicit conversion from one data type to another.

card session

A card session begins when it is powered up or reset. The card is then able to exchange messages with external clients. The card session ends when the card loses power or is reset.

client application

An on-card application that uses services provided by other applications (server applications).

constant pool

The constant pool contains variable-length structures representing various string constants, class names, field names, and other constants referred to within the CAP file and the Export File structure. Each of the constant pool entries, including entry zero, is a variable-length structure whose format is indicated by its first tag byte. There are no ordering constraints on entries in the constant pool entries. One constant pool is associated with each CAP file.

There are differences between the Java platform constant pool and the Java Card technology-based constant pool. For example, in the Java platform 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. The additional information provided by a constant type in Java Card technologies simplifies resolution of references.

context

A protected object space associated with each applet CAP file and Java Card RE. All objects owned by an applet belong to the context associated with the applet’s CAP file.
**context switch**

a change from one currently active context to another. For example, a context switch is caused by an attempt to access an object that belongs to an applet instance that resides in a different context. The result of a context switch is a new currently active context.

**converter**

a piece of software that pre-processes all of the Java programming language class files contained in a set of packages and converts them into a CAP file. The Converter also produces export files for exported packages.

**currently active context**

when an object instance method is invoked, an owning context of this object becomes the currently active context.

**currently selected applet**

the Java Card RE keeps track of the currently selected Java Card applet. Upon receiving a SELECT FILE command with this applet’s AID, the Java Card RE makes this applet the currently selected applet. The Java Card RE sends all APDU commands to the currently selected applet.

**custom CAP file component**

a new component added to the CAP file. The new component must conform to the general component format. It is silently ignored by a Java Card virtual machine that does not recognize the component. The identifiers associated with the new component are recorded in the `custom_component` item of the CAP file’s Directory component.

**D**

**default applet**

an applet that is selected by default on a logical channel in the APDU application environment when it is opened. If an applet is designated the default applet on a particular logical channel in the APDU application environment on the Java Card platform, it becomes the active applet by default when that logical channel is opened using the basic channel.

**E**

**EEPROM**

an acronym for Electrically Erasable, Programmable Read Only Memory.
**entry point method**

well-defined method of an object owned by an application (respectively the Java Card RE) that can be "legally" invoked by another application or the Java Card RE (respectively an application). SIO methods and other container-managed objects' lifecycle methods are application entry point methods. Java Card RE entry point objects' methods are Java Card RE entry point methods.

**entry point objects**

see Java Card RE entry point objects.

**export file**

a file produced by the Converter tool that represents the fields and methods of a package that can be imported by classes in other classic applet applications and classic libraries.

**externally visible**

in the Java Card platform, any classes, interfaces, their constructors, methods and fields that can be accessed from package according to the Java programming language semantics, as defined by the Java Language Specification.

Externally visible items are represented in an export file. For a library package, externally visible items are represented in an export file. For an applet package, only those externally visible items that are part of a shareable interface are represented in an export file.

A Java Card CAP file may restrict the visibility of a package it contains. In this case, these packages are only visible to the other packages inside the CAP file and are not be accessible by packages in other CAP files. No export file is generated for the packages that have their visibility restricted to packages inside the same CAP file.

**finalization**

the process by which a Java virtual machine (VM) allows an unreferenced object instance to release non-memory resources (for example, close and open files) prior to reclaiming the object's memory.

Finalization is only performed on an object when that object is ready to be garbage collected (meaning, there are no references to the object).

Finalization is not supported by the Java Card virtual machine. The method `finalize()` is not called automatically by the Java Card virtual machine.

**firewall**

the mechanism that prevents unauthorized accesses to objects in one application group context from another application group context.
flash memory
a type of persistent mutable memory. It is more efficient in space and power than EPROM. Flash memory can be read bit by bit but can be updated only as a block. Thus, flash memory is typically used for storing additional programs or large chunks of data that are updated as a whole.

framework
the set of classes that implement the API. This includes core and extension packages. Responsibilities include applet selection, sending APDU bytes, and managing atomicity.

garbage collection
the process by which dynamically allocated storage is automatically reclaimed during the execution of a program.

global array
an array objects accessible from any context.

group context
protected object space associated with each CAP file and Java Card RE defining the boundaries of the firewall.

heap
a common pool of free memory in volatile and persistent spaces usable by a program for dynamic memory allocation, in which blocks of memory are used in an arbitrary order. The Java Card virtual machine’s heap is not required to be garbage collected and objects allocated from the heap are not necessarily reclaimed.

installer
the on-card mechanism to download and install CAP files. The installer receives executable binary from the off-card installation program, writes the binary into the smart card memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card Runtime Environment.
installation program
the off-card mechanism that employs a card acceptance device (CAD) to transmit the executable binary in a CAP file to the installer running on the card.

instance variables
also known as non-static fields.

instantiation
in object-oriented programming, 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.

instruction
a statement that indicates an operation for the computer to perform and any data to be used in performing the operation. An instruction can be in machine language or a programming language.

internally visible
code items that are not externally visible. These items are not described in a package's export file and use private tokens to represent internal references. See externally visible.

J

JAR file
an acronym for Java Archive file, which is a file format used for aggregating and compressing many files into one.

Java Card Platform Remote Method Invocation
a subset of the Java Platform Remote Method Invocation (RMI) system optionally supported by the Java Card RE. It provides a mechanism for a client application to invoke a method on a remote object of an applet on the card.

Java Card Runtime Environment (Java Card RE)
consists of the Java Card virtual machine, the application framework, and the associated native methods.

Java Card Virtual Machine (Java Card VM)
a subset of the Java virtual machine, which is designed to be run on smart cards and other resource-constrained devices. The Java Card VM acts an engine that loads Java class files and executes them with a particular set of semantics.
Java Card RE context

the context of the Java Card RE has special system privileges so that it can perform operations that are denied to contexts of applications.

Java Card RE entry point object

an object owned by the Java Card RE context that contains entry point methods. These methods can be invoked from any context and allows applications to request Java Card RE system services. A Java Card RE entry point object can be either temporary or permanent:

- **temporary** - references to temporary Java Card RE entry point objects cannot be stored in class variables, instance variables or array components. The Java Card RE detects and restricts attempts to store references to these objects as part of the firewall functionality to prevent unauthorized reuse. Examples of these objects are APDU objects and the APDU byte array.

- **permanent** - references to permanent Java Card RE entry point objects can be stored and freely reused. Examples of these objects are Java Card RE-owned AID instances.

JDK software

an acronym for Java Development Kit. The JDK software provides the environment required for software development in the Java programming language. The JDK software is available for a variety of operating systems.

library CAP file

a CAP file that contains only library packages. See library package.

library package

a Java programming language package that does not contain any non-abstract classes that extend the class javacard.framework.Applet. See also applet package.

local variable

a data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and cannot be used outside the method.

logical channel

as seen at the card edge, works as a logical link to an applet application on the card. A logical channel establishes a communications session between a card applet and the terminal. Commands issued on a specific logical channel are forwarded to the active applet on that logical channel. For more information, see the ISO/IEC 7816 Specification, Part 4. (http://www.iso.org).
MAC
an acronym for Message Authentication Code. MAC is an encryption of data for security purposes.

mask production (masking)
refers to embedding the Java Card virtual machine, runtime environment, and applications in the read-only memory of a smart card during manufacture.

method
a procedure or routine associated with one or more classes in object-oriented languages.

multiselectable applets
implements the javacard.framework.MultiSelectable interface. Multiselectable applets can be selected on multiple logical channels in the APDU application environment at the same time. They can also accept other applets belonging to the same applet application being selected simultaneously.

multiselected applet
an applet instance that is selected and, therefore, active on more than one logical channel in the APDU application environment simultaneously.

namespace
a set of names in which all names are unique.

native method
a method that is not implemented in the Java programming language, but in another language. The CAP file format does not support native methods to prevent from loading untrusted code.

nibble
four bits.

non-volatile memory
memory that is expected to retain its contents between card tear and power up events or across a reset event on the smart card device.
object-oriented

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.

object owner (Classic)

the applet instance within the currently active context when the object is instantiated. An object can be owned by an applet instance, or by the Java Card RE.

object

in object-oriented programming, unique instance 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.

origin logical channel

the logical channel in the APDU application environment on which an APDU command is issued.

owning context

the application or Java Card RE context in which an object is instantiated or created.

owner context

see owning context.

P

package

a namespace within the Java programming language that can have classes and interfaces.

PCD

an acronym for Proximity Coupling Device. The PCD is a contactless card reader device.

persistent object

persistent objects and their values persist from one card session to the next, indefinitely. Objects are persistent when referred from another persistent object. Persistent object values are typically updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized and deserialized, just that the objects are not lost when the card loses power.
PIX
see AID (application identifier).

RAM (random access memory)
temporary working space for storing and modifying data. RAM is non-persistent memory; that is, the information content is not preserved when power is removed from the memory cell. RAM can be accessed an unlimited number of times and none of the restrictions of EEPROM apply.

reference implementation (RI)
functional and fully compatible implementation of a given technology. It enables developers to build prototypes of applications based on the technology.

remote interface
an interface of an applet application, which extends, directly or indirectly, the java.rmi.Remote interface.

Each method declaration in the remote interface or its super-interfaces includes the exception java.rmi.RemoteException (or one of its super classes) in its throws clause.

In a remote method declaration, if a remote object is declared as a return type, it is declared as the remote interface, not the implementation class of that interface.


remote methods
the methods of a remote interface of an applet application.

remote object
an object of an applet application whose remote methods can be invoked remotely from the off-card client. A remote object is described by one or more remote interfaces of an applet application.

RFU
acronym for Reserved for Future Use.

RID
see AID (application identifier).
RMI
an acronym for Remote Method Invocation. RMI is a mechanism for invoking instance methods on objects located on remote virtual machines (meaning, a virtual machine other than that of the invoker).

ROM (read-only memory)
memory used for storing the fixed program of the card. A smart card's ROM contains operating system routines as well as permanent data and user applications. No power is needed to hold data in this kind of memory. ROM cannot be written to after the card is manufactured. Writing a binary image to the ROM is called masking and occurs during the chip manufacturing process.

runtime environment
see Java Card Runtime Environment (Java Card RE).

S

service
a shareable interface object that a server application uses to provide a set of well-defined functionalities to its clients.

shareable interface
an interface that defines a set of shared methods. These interface methods can be invoked from an application in one context when the object implementing them is owned by an applet in another context.

shareable interface object (SIO)
an object that implements the shareable interface.

smart card
a card that stores and processes information through the electronic circuits embedded in silicon in the substrate of its body. Unlike magnetic stripe cards, smart cards carry both processing power and information. They do not require access to remote databases at the time of a transaction.

SPI
an acronym for Service Provider Interface or sometimes for System Programming Interface. The SPI defines calling conventions by which a platform implementer may implement system services.
terminal
is typically a computer in its own right with an interface which connects with a smart card to exchange and process data.

thread
the basic unit of program execution. A process can have several threads running concurrently each performing a different job, such as waiting for events or performing a time-consuming job that the program doesn't need to complete before going on. When a thread has finished its job, it is suspended or destroyed.

The Java Card virtual machine can support only a single thread of execution. Java Card technology programs cannot use class Thread or any of the thread-related keywords in the Java programming language.

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

transient object
the state of transient objects does not persist from one card session to the next and is reset to a default state at specified intervals. Updates to the values of transient objects are not atomic and are not affected by transactions.

uniform resource identifier (URI)
a compact string of characters used to identify or name an abstract or physical resource. A URI can be further classified as a uniform resource locator (URL), a uniform resource name (URN), or both. See RFC 3986 for more information.

uniform resource locator (URL)
a compact string representation used to locate resources available via network protocols or other protocols. Once the resource represented by a URL has been accessed, various operations may be performed on that resource. See RFC 1738 for more information. A URL is a type of uniform resource identifier (URI).
**V**

verification

a process performed on an application or library executable that checks that the binary representation of the application or library is structurally correct and type safe.

volatile memory

memory that is not expected to retain its contents between card tear and power up events or across a reset event on the smart card device.

volatile object

an object that is ideally suited to be stored in volatile memory. This type of object is intended for a short-lived object or an object, which requires frequent updates. A volatile object is garbage collected on card tear (or reset).

**W**

word

an abstract storage unit. 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 integer type.
Annex A - Oracle Technology Network Developer License Terms

Specifications

Export Controls

Export laws and regulations of the United States and any other relevant local export laws and regulations apply to the specifications. You agree that such export control laws govern your use of the specifications (including technical data), and you agree to comply with all such export laws and regulations (including "deemed export" and "deemed re-export" regulations). You agree that no data, information, program and/or materials resulting from services (or direct product thereof) will be exported, directly or indirectly, in violation of these laws, or will be used for any purpose prohibited by these laws including, without limitation, nuclear, chemical, or biological weapons proliferation, or development of missile technology.

Accordingly, you confirm:

- You will not download, provide, make available or otherwise export or re-export the specifications, directly or indirectly, to countries prohibited by applicable laws and regulations nor to citizens, nationals or residents of those countries.

- You are not listed on the United States Department of Treasury lists of Specially Designated Nationals and Blocked Persons, Specially Designated Terrorists, and Specially Designated Narcotic Traffickers, nor are you listed on the United States Department of Commerce Table of Denial Orders.

- You will not download or otherwise export or re-export the specifications, directly or indirectly, to persons on the above mentioned lists.

- You will not use the specifications for, and will not allow the specifications to be used for, any purposes prohibited by applicable law, including, without limitation, for the development, design, manufacture or production of nuclear, chemical or biological weapons of mass destruction.

Oracle Employees: Under no circumstances are Oracle Employees authorized to download the specifications for the purpose of distributing it to customers. Oracle products are available to employees for internal use or demonstration purposes only. In keeping with Oracle's trade
compliance obligations under U.S. and applicable multilateral law, failure to comply with this policy could result in disciplinary action up to and including termination.

**PLEASE READ THE FOLLOWING LICENSE AGREEMENT TERMS AND CONDITIONS CAREFULLY BEFORE INSTALLING OR USING THE SPECIFICATIONS. THESE TERMS AND CONDITIONS CONSTITUTE A LEGAL AGREEMENT BETWEEN YOU AND ORACLE.**

**ORACLE TECHNOLOGY NETWORK LICENSE AGREEMENT**

"We," "us," and "our" refers to Oracle America, Inc., for and on behalf of itself and its subsidiaries and affiliates under common control. "You" and "your" refers to the individual or entity that wishes to use the specification from Oracle. "Specifications" refers to the Java Card Classic Edition specification document and/or Java Card Connected Edition specification document that you selected for download or use from Oracle and any other Oracle product or technology documentation provided to you by Oracle under this agreement. "License" refers to your right to use the specifications under the terms of this agreement. “Applications” means Java technology applications intended to run on the Java Card Classic and/or Java Card Connected platforms. This agreement is governed by the substantive and procedural laws of California. You and Oracle agree to submit to the exclusive jurisdiction of, and venue in, the courts of San Francisco or Santa Clara counties in California in any dispute arising out of or relating to this agreement.

We are willing to license the specifications to you only upon the condition that you accept all of the terms contained in this agreement. Read the terms carefully and select the "Accept License Agreement" button to confirm your acceptance. If you are not willing to be bound by these terms, select the "Decline License Agreement" button and the registration process will not continue.

**LICENSE RIGHTS**

Except for any included software package or file that is licensed to you by Oracle under different license terms, we grant you a perpetual (unless terminated as provided in this agreement), nonexclusive, nontransferable, limited License to use (without the right to sublicense) the specifications internally solely for the purposes of designing and developing your implementation of the specifications and designing and developing your applets and applications intended to run on the Java Card platform. Other than this limited license, you acquire no right, title or interest in or to the specifications or any other Oracle intellectual property. You acknowledge that any commercial or productive use of an implementation of the
specifications requires separate and appropriate licensing agreements.

All rights not expressly granted above are hereby reserved. If you want to use the specifications for any purpose other than as permitted under this agreement, including but not limited to distribution of the specifications or any use of the specifications for your internal business purposes (other than developing, testing, prototyping and demonstrating your applications) or for any commercial production purposes, you must obtain a valid license permitting such use. We may audit your use of the specifications.

Third-Party Technology

The specifications may contain or be distributed with certain third-party technology. Oracle may provide certain notices related to such third-party technology in the specifications.

Third party technology will be licensed to you either under the terms of this agreement, or, if specified in the specifications, under separate license terms ("Separate Terms") and not under the terms of this agreement ("Separately Licensed Third Party Technology"). Licensee's rights to use such Separately Licensed Third Party Technology under the Separate Terms are not restricted or modified in any way by this Agreement.

Ownership and Restrictions

We retain all ownership and intellectual property rights in the specifications. Unless enforcement is prohibited by applicable law, you may not modify the specifications. You may make a sufficient number of copies of the specifications for the licensed use and one copy of the specifications for backup purposes.

You may not:
- use the specifications for your own internal business purposes (other than developing, testing, prototyping and demonstrating your applications) or for any commercial or production purposes;

- remove or modify any program markings or any notice of our proprietary rights;

- make the specifications available in any manner to any third party;

- use the specifications to provide third party training;

- assign this agreement or give or transfer the specifications or an interest in them to another
individual or entity;
- cause or permit modification of the specifications;
- disclose results of any benchmark test results related to the specifications without our prior consent.
- create, modify, or change the behavior of classes, interfaces, or subpackages that are in any way identified as "java", "javafx", "javae", "sun", "oracle" or similar convention as specified by Oracle in any naming convention designation;

**Export**

You agree that U.S. export control laws and other applicable export and import laws govern your use of the specifications, including technical data; additional information can be found on Oracle's Global Trade Compliance web site located at https://www.oracle.com/products/export-regulations.html. You agree that neither the specifications nor any direct product thereof will be exported, directly, or indirectly, in violation of these laws, or will be used for any purpose prohibited by these laws including, without limitation, nuclear, chemical, or biological weapons proliferation.

**Disclaimer of Warranty and Exclusive Remedies**

THE SPECIFICATIONS IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. WE FURTHER DISCLAIM ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT.

IN NO EVENT SHALL WE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. OUR ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).
No Technical Support

Our technical support organization will not provide technical support, phone support, or updates to you for the specifications licensed under this agreement.

End of Agreement

You may terminate this agreement by destroying all copies of the specifications. We have the right to terminate your right to use the specifications if you fail to comply with any of the terms of this agreement, in which case you shall destroy all copies of the specifications.

Relationship Between the Parties

The relationship between you and us is that of licensee/licensor. Neither party will represent that it has any authority to assume or create any obligation, express or implied, on behalf of the other party, nor to represent the other party as agent, employee, franchisee, or in any other capacity. Nothing in this agreement shall be construed to limit either party's right to independently develop or distribute software that is functionally similar to the other party's products, so long as proprietary information of the other party is not included in such software.

Open Source Software

"Open Source" software - software available without charge for use, modification and distribution - is often licensed under terms that require the user to make the user's modifications to the Open Source software or any software that the user 'combines' with the Open Source software freely available in source code form. If you use Open Source software in conjunction with the specifications, you must ensure that your use does not: (i) create, or purport to create, obligations of us with respect to the Oracle specifications; or (ii) grant, or purport to grant, to any third party any rights to or immunities under our intellectual property or proprietary rights in the Oracle specifications. For example, you may not develop a software program using an Oracle program/specification and an Open Source program where such use results in a program file(s) that contains code from both the Oracle program/specification and the Open Source program (including without limitation libraries) if the Open Source program is licensed under a license that requires any "modifications" be made freely available. You also may not combine the Oracle specifications with a program licensed under the GNU General Public License ("GPL") in any manner that could cause, or could be interpreted or asserted to cause, the Oracle specifications or any modifications thereto to become subject to the terms of the GPL.
Entire Agreement

You agree that this agreement is the complete agreement for the specifications and licenses, and this agreement supersedes all prior or contemporaneous agreements or representations. If any term of this agreement is found to be invalid or unenforceable, the remaining provisions will remain effective.

Last updated: 3 April 2012

Should you have any questions concerning this License Agreement, or if you desire to contact Oracle for any reason, please write:

Oracle America, Inc.
500 Oracle Parkway, Redwood City, CA 94065