The Java™ Series

The Java™ Programming Language
Ken Arnold and James Gosling
ISBN 0-201-63455-4

The Java™ Language Specification
James Gosling, Bill Joy, and Guy Steele
ISBN 0-201-63451-1

The Java™ Virtual Machine Specification
Tim Lindholm and Frank Yellin
ISBN 0-201-63452-X

The Java™ Application Programming Interface, Volume 1: Core Packages
James Gosling, Frank Yellin, and the Java Team
ISBN 0-201-63453-8

The Java™ Application Programming Interface, Volume 2: Window Toolkit and Applets
James Gosling, Frank Yellin, and the Java Team
ISBN 0-201-63459-7

The Java™ Tutorial: Object-Oriented Programming for the Internet
Mary Campione and Kathy Walrath
ISBN 0-201-63454-6

The Java™ Class Libraries: An Annotated Reference
Patrick Chan and Rosanna Lee
ISBN 0-201-63458-9

The Java™ FAQ: Frequently Asked Questions
Jonni Kanerva
LIMITED LICENSE GRANTS

1. Sun Microsystems, Inc. ("Sun") hereby grants to you with respect to the Java Virtual Machine Specification, Third Edition ("Specification"), under any applicable copyrights or, subject to the provisions of subsection 3 below, patent rights Sun may have covering the Specification, the following perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited licenses (without the right to sublicense):

   i. to use the Java Virtual Machine Specification, Third Edition ("Specification") for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Specification; and

   ii. to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties). Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

This License will terminate immediately without notice from Sun if you breach the License or act outside the scope of the licenses granted above.

For the purposes of this License: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Sun's source code or binary code materials nor, except with an appropriate and separate license from Sun, includes any of Sun's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.sun" or their equivalents in any subsequent naming convention adopted by Sun through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Sun which corresponds to the Specification and that was available either (i) from Sun's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

2. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Sun's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

b. With respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Sun that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c. Also with respect to any patent claims owned by Sun and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Sun that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

DISCLAIMER OF WARRANTIES

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES MAY BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

OTHER TERMS

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

Sun, Sun Microsystems, the Sun logo, Java, HotJava, JDK, JVM, and all Java-based trademarks or logos are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX® is a registered trademark of The Open Group in the United States and other countries. All other product names mentioned herein are the trademarks of their respective owners.

Duke logo™ designed by Joe Palrang.
To Lucy, Beatrice, and Arnold —TL

To Weihong and Teva —GB

To Mark —FY
# The class File Format

4.1 The ClassFile Structure ........................................... 42
4.2 The Internal Form of Names ...................................... 48
  4.2.1 Binary Class and Interface Names .......................... 48
  4.2.2 Unqualified Names ........................................... 48
4.3 Descriptors and Signatures ...................................... 48
  4.3.1 Grammar Notation ............................................ 49
  4.3.2 Field Descriptors ........................................... 49
  4.3.3 Method Descriptors ......................................... 51
  4.3.4 Signatures ................................................... 52
4.4 The Constant Pool ................................................ 55
  4.4.1 The CONSTANT_Class_info Structure ....................... 56
  4.4.2 The CONSTANT_Fieldref_info, CONSTANT_Methodref_info, and CONSTANT_InterfaceMethodref_info Structures ................................. 57
  4.4.3 The CONSTANT_String_info Structure ..................... 59
  4.4.4 The CONSTANT_Integer_info and CONSTANT_Float_info Structures ................................................................. 59
  4.4.5 The CONSTANT_Long_info and CONSTANT_Double_info Structures ................................................................. 61
  4.4.6 The CONSTANT_NameAndType_info Structure ............... 62
  4.4.7 The CONSTANT_Utf8_info Structure ....................... 63
4.5 Fields .................................................................... 65
4.6 Methods .................................................................. 68
4.7 Attributes ............................................................. 72
  4.7.1 Defining and Naming New Attributes ....................... 74
  4.7.2 The ConstantValue Attribute ................................ 74
  4.7.3 The Code Attribute .......................................... 76
  4.7.4 The StackMapTable Attribute ................................ 80
  4.7.5 The Exceptions Attribute .................................. 88
  4.7.6 The InnerClasses Attribute ................................ 89
  4.7.7 The EnclosingMethod Attribute .............................. 92
  4.7.8 The Synthetic Attribute .................................... 94
  4.7.9 The Signature Attribute .................................... 94
  4.7.10 The SourceFile Attribute .................................. 96
  4.7.11 The SourceDebugExtension Attribute .................... 97
  4.7.12 The LineNumberTable Attribute ........................... 97
  4.7.13 The LocalVariableTable Attribute ....................... 99
  4.7.14 The LocalVariableTypeInfo Table Attribute .......... 101
  4.7.15 The Deprecated Attribute .................................. 103
  4.7.16 The RuntimeVisibleAnnotations attribute ............. 104
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7.16.1</td>
<td>The element_value structure</td>
<td>106</td>
</tr>
<tr>
<td>4.7.17</td>
<td>The RuntimeInvisibleAnnotations attribute</td>
<td>109</td>
</tr>
<tr>
<td>4.7.18</td>
<td>The RuntimeVisibleParameterAnnotations attribute</td>
<td>110</td>
</tr>
<tr>
<td>4.7.19</td>
<td>The RuntimeInvisibleParameterAnnotations attribute</td>
<td>112</td>
</tr>
<tr>
<td>4.7.20</td>
<td>The AnnotationDefault attribute</td>
<td>113</td>
</tr>
<tr>
<td>4.8</td>
<td>Format Checking</td>
<td>115</td>
</tr>
<tr>
<td>4.9</td>
<td>Constraints on Java Virtual Machine Code</td>
<td>115</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Static Constraints</td>
<td>115</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Structural Constraints</td>
<td>119</td>
</tr>
<tr>
<td>4.10</td>
<td>Verification of class Files</td>
<td>122</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Verification by Type Checking</td>
<td>123</td>
</tr>
<tr>
<td>4.10.1.1</td>
<td>The Type Hierarchy</td>
<td>128</td>
</tr>
<tr>
<td>4.10.1.2</td>
<td>Subtyping Rules</td>
<td>130</td>
</tr>
<tr>
<td>4.10.1.3</td>
<td>Typechecking Rules</td>
<td>134</td>
</tr>
<tr>
<td>4.10.1.4</td>
<td>Instructions</td>
<td>147</td>
</tr>
<tr>
<td>4.10.2</td>
<td>Verification by Type Inference</td>
<td>226</td>
</tr>
<tr>
<td>4.10.2.1</td>
<td>The Process of Verification by Type Inference</td>
<td>226</td>
</tr>
<tr>
<td>4.10.2.2</td>
<td>The Bytecode Verifier</td>
<td>227</td>
</tr>
<tr>
<td>4.10.2.3</td>
<td>Values of Types long and double</td>
<td>230</td>
</tr>
<tr>
<td>4.10.2.4</td>
<td>Instance Initialization Methods and Newly Created Objects</td>
<td>230</td>
</tr>
<tr>
<td>4.11</td>
<td>Limitations of the Java Virtual Machine</td>
<td>234</td>
</tr>
<tr>
<td>5</td>
<td>Loading, Linking, and Initializing.</td>
<td>237</td>
</tr>
<tr>
<td>5.1</td>
<td>The Runtime Constant Pool</td>
<td>237</td>
</tr>
<tr>
<td>5.2</td>
<td>Virtual Machine Start-up</td>
<td>240</td>
</tr>
<tr>
<td>5.3</td>
<td>Creation and Loading</td>
<td>240</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Loading Using the Bootstrap Class Loader</td>
<td>242</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Loading Using a User-defined Class Loader</td>
<td>242</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Creating Array Classes</td>
<td>243</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Loading Constraints</td>
<td>244</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Deriving a Class from a class File Representation</td>
<td>245</td>
</tr>
<tr>
<td>5.4</td>
<td>Linking</td>
<td>247</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Verification</td>
<td>247</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Preparation</td>
<td>248</td>
</tr>
<tr>
<td>5.4.2.1</td>
<td>Method overriding</td>
<td>248</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Resolution</td>
<td>249</td>
</tr>
<tr>
<td>5.4.3.1</td>
<td>Class and Interface Resolution</td>
<td>250</td>
</tr>
<tr>
<td>5.4.3.2</td>
<td>Field Resolution</td>
<td>250</td>
</tr>
<tr>
<td>5.4.3.3</td>
<td>Method Resolution</td>
<td>251</td>
</tr>
<tr>
<td>5.4.3.4</td>
<td>Interface Method Resolution</td>
<td>252</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Access Control</td>
<td>252</td>
</tr>
<tr>
<td>5.5</td>
<td>Initialization</td>
<td>253</td>
</tr>
<tr>
<td>5.6</td>
<td>Binding Native Method Implementations</td>
<td>255</td>
</tr>
<tr>
<td>5.7</td>
<td>Virtual Machine Exit</td>
<td>256</td>
</tr>
<tr>
<td>6</td>
<td>The Java Virtual Machine Instruction Set</td>
<td>257</td>
</tr>
<tr>
<td>6.1</td>
<td>Assumptions: The Meaning of “Must”</td>
<td>257</td>
</tr>
<tr>
<td>6.2</td>
<td>Reserved Opcodes</td>
<td>258</td>
</tr>
<tr>
<td>6.3</td>
<td>Virtual Machine Errors</td>
<td>258</td>
</tr>
</tbody>
</table>
6.4 Format of Instruction Descriptions ............................................. 259

7 Compiling for the Java Virtual Machine ......................... 449
  7.1 Format of Examples .................................................. 450
  7.2 Use of Constants, Local Variables, and Control Constructs .... 451
  7.3 Arithmetic .................................................................... 456
  7.4 Accessing the Runtime Constant Pool ............................ 457
  7.5 More Control Examples ............................................. 458
  7.6 Receiving Arguments ................................................. 462
  7.7 Invoking Methods ..................................................... 463
  7.8 Working with Class Instances ..................................... 465
  7.9 Arrays ......................................................................... 468
  7.10 Compiling Switches ...................................................... 471
  7.11 Operations on the Operand Stack ................................. 473
  7.12 Throwing and Handling Exceptions ......................... 474
  7.13 Compiling finally ......................................................... 479
  7.14 Annotations .............................................................. 482
  7.15 Synchronization .......................................................... 482

8 Threads and Locks ............................................................... 485

9 Opcode Mnemonics by Opcode ................................. 487

Index ................................................................. 491
THE Java™ virtual machine specification has been written to fully document the design of the Java virtual machine. It is essential for compiler writers who wish to target the Java virtual machine and for programmers who want to implement a compatible Java virtual machine. It is also a definitive source for anyone who wants to know exactly how the Java programming language is implemented.

The Java virtual machine is an abstract machine. References to the Java virtual machine throughout this specification refer to this abstract machine rather than to Sun’s or any other specific implementation. This book serves as documentation for a concrete implementation of the Java virtual machine only as a blueprint documents a house. An implementation of the Java virtual machine (known as a runtime interpreter) must embody this specification, but is constrained by it only where absolutely necessary.


We intend that this specification should sufficiently document the Java virtual machine to make possible compatible clean-room implementations. If you are considering constructing your own Java virtual machine implementation, feel free to contact us to obtain assistance to ensure the 100% compatibility of your implementation.

Send comments on this specification or questions about implementing the Java virtual machine to our electronic mail address: jvm@java.sun.com. To learn the latest about the Java Platform, or to download the latest JDK™ release, visit our World Wide Web site at http://java.sun.com. For updated information about the Java Series, including errata for The Java™ Virtual Machine Specification, and previews of forthcoming books, visit http://java.sun.com/Series.

The virtual machine that evolved into the Java virtual machine was originally designed by James Gosling in 1992 to support the Oak programming language. The evolution into its present form occurred through the direct and indirect efforts
of many people and spanned Sun’s Green project, FirstPerson, Inc., the LiveOak project, the Java Products Group, JavaSoft, and today, Sun’s Client Platform Group. The authors are grateful to the many contributors and supporters.

This book began as internal project documentation. Kathy Walrath edited that early draft, helping to give the world its first look at the internals of the Java programming language. It was then converted to HTML by Mary Campione and was made available on our Web site before being expanded into book form.

The creation of *The Java™ Virtual Machine Specification* owes much to the support of the Java Products Group led by General Manager Ruth Hennigar, to the efforts of series editor Lisa Friendly, and to editor Mike Hendrickson and his group at Addison-Wesley. The many criticisms and suggestions received from reviewers of early online drafts, as well as drafts of the printed book, improved its quality immensely. We owe special thanks to Richard Tuck for his careful review of the manuscript. Particular thanks to Bill Joy whose comments, reviews, and guidance have contributed greatly to the completeness and accuracy of this book.

### Notes on the Second Edition

The second edition of *The Java™ Virtual Machine Specification* brings the specification of the Java virtual machine up to date with the Java® 2 platform, v1.2. It also includes many corrections and clarifications that update the presentation of the specification without changing the logical specification itself. We have attempted to correct typos and errata (hopefully without introducing new ones) and to add more detail to the specification where it was vague or ambiguous. In particular, we corrected a number of inconsistencies between the first edition of *The Java™ Virtual Machine Specification* and *The Java™ Language Specification*.

We thank the many readers who combed through the first edition of this book and brought problems to our attention. Several individuals and groups deserve special thanks for pointing out problems or contributing directly to the new material:

Carla Schroer and her teams of compatibility testers in Cupertino, California, and Novosibirsk, Russia (with special thanks to Leonid Arbouzov and Alexei Kaigorodov), painstakingly wrote compatibility tests for each testable assertion in the first edition. In the process they uncovered many places where the original specification was unclear or incomplete.
Jeroen Vermeulen, Janice Shepherd, Peter Bertelsen, Roly Perera, Joe Darcy, and Sandra Loosemore have all contributed comments and feedback that have improved this edition.

Marilyn Rash and Hilary Selby Polk of Addison Wesley Longman helped us to improve the readability and layout of this edition at the same time as we were incorporating all the technical changes.

Special thanks go to Gilad Bracha, who has brought a new level of rigor to the presentation and has been a major contributor to much of the new material, especially chapters 4 and 5. His dedication to “computational theology” and his commitment to resolving inconsistencies between The Java™ Virtual Machine Specification and The Java™ Language Specification have benefited this book tremendously.

*Tim Lindholm*

*Frank Yellin*

Java Software, Sun Microsystems, Inc

---

**Notes on the Third Edition**

The third edition of The Java™ Virtual Machine Specification incorporates all the changes that have been made to the specification through Java SE 6. Chief among these is the new approach to bytecode verification. In addition, numerous corrections and clarifications that have been made over time. Changes that have been added to support new features of the Java programming language are integrated herein as well.

More people than we can mention here have, over time, contributed to the design and implementation of the Java virtual machine. It’s worth belatedly mentioning some highlights nevertheless.

The excellent performance we see in the JVMs of today would never have been possible without the technological foundation laid by David Ungar and his colleagues at the Self project at Sun Labs. This technology took a convoluted path, from Self on through the Animorphic Smalltalk VM to eventually become Sun’s Hotspot virtual machine. Lars Bak and Urs Hoelzle are the two people who were present through all these stages, and are more responsible than anyone else for the high performance we take for granted in JVMs today.
At the same time, JVMs have also become very small. The KVM, designed for mobile devices, was pioneered by Antero Taivalsaari. The new verification technique specified in this edition was pioneered in the KVM context.

It was Eva Rose who, in her masters thesis, first proposed a radical revision of JVM bytecode verification, in the context of the Java Card™ platform. This led first to an implementation for Java ME CLDC, and eventually to the revision of the Java SE verification process documented here.

Sheng Liang implemented the Java ME CLDC verifier. I was responsible for specifying the verifier, and Antero Taivalsaari led the overall specification of Java ME CLDC. Alessandro Coglio’s analysis of Java bytecode verification was the most extensive, realistic and thorough study of the topic, and contributed greatly to this specification.

Wei Tao, together with Frank Yellin, Tim Lindholm and myself, implemented the Prolog verifier that formed the basis for the specification in both Java ME and Java SE. Wei then implemented the specification "for real" in the JVM. Later, Mingyao Yang improved the design and specification, and implemented the final version that ships in Java SE 6.

This specification benefited from the efforts of the JSR 202 expert group: Peter Burka, Alessandro Coglio, Sanghoon Jin, Christian Kemper, Larry Rau, Eva Rose and Mark Stolz.

My director, Janet Koenig, did everything she possibly could to provide an excellent working environment. Brian Goetz and David Holmes provided valuable feedback and discussion.

Above all, I wish to acknowledge the special contribution of Alex Buckley, my successor at Sun, who did all the hard work to actually get this book out the door. Alex made significant improvements to the text, especially in chapter 4. He reviewed the book as a whole and helped make it more consistent and cohesive; any remaining inconsistencies and errors are entirely my fault. Last but not least, Alex prepared this volume for publication. It is a thankless task for which I thank him nonetheless. I know that future revisions of the core Java specifications are in good hands.

Gilad Bracha
Java SE, Sun Microsystems, Inc
CHAPTER 1

Introduction

1.1 A Bit of History

THE Java™ programming language is a general-purpose object-oriented concurrent language. Its syntax is similar to C and C++, but it omits many of the features that make C and C++ complex, confusing, and unsafe. The Java platform was initially developed to address the problems of building software for networked consumer devices. It was designed to support multiple host architectures and to allow secure delivery of software components. To meet these requirements, compiled code had to survive transport across networks, operate on any client, and assure the client that it was safe to run.

The popularization of the World Wide Web made these attributes much more interesting. Web browsers enabled millions of people to surf the Net and access media-rich content in simple ways. At last there was a medium where what you saw and heard was essentially the same regardless of the machine you were using and whether it was connected to a fast network or a slow modem.

Web enthusiasts soon discovered that the content supported by the Web’s HTML document format was too limited. HTML extensions, such as forms, only highlighted those limitations, while making it clear that no browser could include all the features users wanted. Extensibility was the answer.

Sun’s HotJava™ browser first showcased the interesting properties of the Java programming language and platform by making it possible to embed programs inside HTML pages. Programs are transparently downloaded into the browser along with the HTML pages in which they appear. Before being accepted by the browser, programs are carefully checked to make sure they are safe. Like HTML pages, compiled programs are network- and host-independent. The programs behave the same way regardless of where they come from or what kind of machine they are being loaded into and run on.
A Web browser incorporating the Java platform is no longer limited to a pre-determined set of capabilities. Visitors to Web pages incorporating dynamic content can be assured that their machines cannot be damaged by that content. Programmers can write a program once, and it will run on any machine supplying a Java runtime environment.

1.2 The Java Virtual Machine

The Java virtual machine is the cornerstone of the Java platform. It is the component of the technology responsible for its hardware- and operating system-independence, the small size of its compiled code, and its ability to protect users from malicious programs.

The Java virtual machine is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at runtime. It is reasonably common to implement a programming language using a virtual machine; the best-known virtual machine may be the P-Code machine of UCSD Pascal.

The first prototype implementation of the Java virtual machine, done at Sun Microsystems, Inc., emulated the Java virtual machine instruction set in software hosted by a handheld device that resembled a contemporary Personal Digital Assistant (PDA). Sun’s current implementations emulate the Java virtual machine on mobile, desktop and server devices, but the Java virtual machine does not assume any particular implementation technology, host hardware, or host operating system. It is not inherently interpreted, but can just as well be implemented by compiling its instruction set to that of a silicon CPU. It may also be implemented in microcode or directly in silicon.

The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the class file format. A class file contains Java virtual machine instructions (or bytecodes) and a symbol table, as well as other ancillary information.

For the sake of security, the Java virtual machine imposes strong syntactic and structural constraints on the code in a class file. However, any language with functionality that can be expressed in terms of a valid class file can be hosted by the Java virtual machine. Attracted by a generally available, machine-independent platform, implementors of other languages can turn to the Java virtual machine as a delivery vehicle for their languages.
1.3 Summary of Chapters

The rest of this book is structured as follows:


- Chapter 3 gives an overview of the Java virtual machine architecture.

- Chapter 4 specifies the class file format, the hardware- and operating system-independent binary format used to represent compiled classes and interfaces.

- Chapter 5 specifies the start-up of the Java virtual machine and the loading, linking, and initialization of classes and interfaces.

- Chapter 6 specifies the instruction set of the Java virtual machine, presenting the instructions in alphabetical order of opcode mnemonics.

- Chapter 7 introduces compilation of code written in the Java programming language into the instruction set of the Java virtual machine.


- Chapter 9 gives a table of Java virtual machine opcode mnemonics indexed by opcode value.

1.4 Notation

Throughout this book we refer to classes and interfaces drawn from the Java platform. Whenever we refer to a class or interface using a single identifier \( N \), the intended reference is to the class or interface \( \text{java.lang.}N \). We use the fully qualified name for classes from packages other than \( \text{java.lang.} \)

Whenever we refer to a class or interface that is declared in the package \( \text{java} \) or any of its subpackages, the intended reference is to that class or interface as loaded by the bootstrap class loader (§5.3.1). Whenever we refer to a subpackage of a package named \( \text{java} \), the intended reference is to that subpackage as determined by the bootstrap class loader.

The use of fonts in this book is as follows:
• A fixed width font is used for code examples written in the Java programming language, Java virtual machine data types, exceptions, and errors.

• *Italic* is used for Java virtual machine “assembly language”, its opcodes and operands, as well as items in the Java virtual machine’s runtime data areas. It is also used to introduce new terms and simply for emphasis.
Java Programming Language Concepts

The Java virtual machine was designed to support the Java programming language. Some concepts and vocabulary from the Java programming language are thus useful when attempting to understand the virtual machine.

In *The Java™ Virtual Machine Specification, Second Edition*, Chapter 2 gave an overview of the Java programming language intended to support the specification of the Java virtual machine, but was not itself a part of that specification.

In this specification, the reader is referred to *The Java™ Language Specification, Third Edition*, by James Gosling, Bill Joy, Guy Steele and Gilad Bracha, for information about the Java programming language. References of the form:

(JLS3 §x.y)

indicate where this is necessary.

For an introduction to the Java programming language, see *The Java™ Programming Language, Fourth Edition*, by Ken Arnold, James Gosling, and David Holmes.
This book specifies an abstract machine. It does not document any particular implementation of the Java virtual machine, including Sun Microsystems’.

To implement the Java virtual machine correctly, you need only be able to read the class file format and correctly perform the operations specified therein. Implementation details that are not part of the Java virtual machine’s specification would unnecessarily constrain the creativity of implementors. For example, the memory layout of runtime data areas, the garbage-collection algorithm used, and any internal optimization of the Java virtual machine instructions (for example, translating them into machine code) are left to the discretion of the implementor.

3.1 The class File Format

Compiled code to be executed by the Java virtual machine is represented using a hardware- and operating system-independent binary format, typically (but not necessarily) stored in a file, known as the class file format. The class file format precisely defines the representation of a class or interface, including details such as byte ordering that might be taken for granted in a platform-specific object file format.

Chapter 4, “The class File Format”, covers the class file format in detail.

3.2 Data Types

Like the Java programming language, the Java virtual machine operates on two kinds of types: primitive types and reference types. There are, correspondingly, two
kinds of values that can be stored in variables, passed as arguments, returned by methods, and operated upon: primitive values and reference values.

The Java virtual machine expects that nearly all type checking is done prior to runtime, typically by a compiler, and does not have to be done by the Java virtual machine itself. Values of primitive types need not be tagged or otherwise be inspectable to determine their types at runtime, or to be distinguished from values of reference types. Instead, the instruction set of the Java virtual machine distinguishes its operand types using instructions intended to operate on values of specific types. For instance, \texttt{iadd}, \texttt{ladd}, \texttt{fadd}, and \texttt{dadd} are all Java virtual machine instructions that add two numeric values and produce numeric results, but each is specialized for its operand type: \texttt{int}, \texttt{long}, \texttt{float}, and \texttt{double}, respectively. For a summary of type support in the Java virtual machine instruction set, see §3.11.1.

The Java virtual machine contains explicit support for objects. An object is either a dynamically allocated class instance or an array. A reference to an object is considered to have Java virtual machine type \texttt{reference}. Values of type \texttt{reference} can be thought of as pointers to objects. More than one reference to an object may exist. Objects are always operated on, passed, and tested via values of type \texttt{reference}.

### 3.3 Primitive Types and Values

The primitive data types supported by the Java virtual machine are the numeric types, the \texttt{boolean} type (§3.3.4),\footnote{The first edition of The Java™ Virtual Machine Specification did not consider \texttt{boolean} to be a Java virtual machine type. However, \texttt{boolean} values do have limited support in the Java virtual machine. The second edition clarified the issue by treating \texttt{boolean} as a type.} and the \texttt{returnAddress} type (§3.3.3). The numeric types consist of the integral types (§3.3.1) and the floating-point types (§3.3.2). The integral types are:

- \texttt{byte}, whose values are 8-bit signed two’s-complement integers
- \texttt{short}, whose values are 16-bit signed two’s-complement integers
- \texttt{int}, whose values are 32-bit signed two’s-complement integers
- \texttt{long}, whose values are 64-bit signed two’s-complement integers
- \texttt{char}, whose values are 16-bit unsigned integers representing UTF-16 code units (JLS3 §3.1)
The floating-point types are:

- `float`, whose values are elements of the float value set or, where supported, the float-extended-exponent value set
- `double`, whose values are elements of the double value set or, where supported, the double-extended-exponent value set

The values of the `boolean` type encode the truth values `true` and `false`.

The values of the `returnAddress` type are pointers to the opcodes of Java virtual machine instructions. Of the primitive types only the `returnAddress` type is not directly associated with a Java programming language type.

### 3.3.1 Integral Types and Values

The values of the integral types of the Java virtual machine are the same as those for the integral types of the Java programming language (JLS3 §4.2.1):

- For `byte`, from $-128$ to $127$ ($-2^7$ to $2^7 - 1$), inclusive
- For `short`, from $-32768$ to $32767$ ($-2^{15}$ to $2^{15} - 1$), inclusive
- For `int`, from $-2147483648$ to $2147483647$ ($-2^{31}$ to $2^{31} - 1$), inclusive
- For `long`, from $-9223372036854775808$ to $9223372036854775807$ ($-2^{63}$ to $2^{63} - 1$), inclusive
- For `char`, from 0 to 65535 inclusive

### 3.3.2 Floating-Point Types, Value Sets, and Values

The floating-point types are `float` and `double`, which are conceptually associated with the 32-bit single-precision and 64-bit double-precision format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985 (IEEE, New York).

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also positive and negative zeros, positive and negative *infinities*, and a special *Not-a-Number value* (hereafter abbreviated as “NaN”). The NaN value is used to represent the result of certain invalid operations such as dividing zero by zero.

Every implementation of the Java virtual machine is required to support two standard sets of floating-point values, called the `float value set` and the `double`
value set. In addition, an implementation of the Java virtual machine may, at its option, support either or both of two extended-exponent floating-point value sets, called the float-extended-exponent value set and the double-extended-exponent value set. These extended-exponent value sets may, under certain circumstances, be used instead of the standard value sets to represent the values of type float or double.

The finite nonzero values of any floating-point value set can all be expressed in the form \( s \cdot m \cdot 2^{(e - N + 1)} \), where \( s \) is +1 or -1, \( m \) is a positive integer less than \( 2^N \), and \( e \) is an integer between \( E_{\text{min}} = -(2^{K-1} - 2) \) and \( E_{\text{max}} = 2^{K-1} - 1 \), inclusive, and where \( N \) and \( K \) are parameters that depend on the value set. Some values can be represented in this form in more than one way; for example, supposing that a value \( v \) in a value set might be represented in this form using certain values for \( s \), \( m \), and \( e \), then if it happened that \( m \) were even and \( e \) were less than \( 2^{K-1} \), one could halve \( m \) and increase \( e \) by 1 to produce a second representation for the same value \( v \). A representation in this form is called normalized if \( m \geq 2^{N-1} \); otherwise the representation is said to be denormalized. If a value in a value set cannot be represented in such a way that \( m \geq 2^{N-1} \), then the value is said to be a denormalized value, because it has no normalized representation.

The constraints on the parameters \( N \) and \( K \) (and on the derived parameters \( E_{\text{min}} \) and \( E_{\text{max}} \)) for the two required and two optional floating-point value sets are summarized in Table 3.1.

### Table 3.1 Floating-point value set parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>float</th>
<th>float-extended-exponent</th>
<th>double</th>
<th>double-extended-exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>24</td>
<td>24</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>( K )</td>
<td>8</td>
<td>( \geq 11 )</td>
<td>11</td>
<td>( \geq 15 )</td>
</tr>
<tr>
<td>( E_{\text{max}} )</td>
<td>+127</td>
<td>( \geq +1023 )</td>
<td>+1023</td>
<td>( \geq +16383 )</td>
</tr>
<tr>
<td>( E_{\text{min}} )</td>
<td>-126</td>
<td>( \leq -1022 )</td>
<td>-1022</td>
<td>( \leq -16382 )</td>
</tr>
</tbody>
</table>

Where one or both extended-exponent value sets are supported by an implementation, then for each supported extended-exponent value set there is a specific implementation-dependent constant \( K \), whose value is constrained by Table 3.1; this value \( K \) in turn dictates the values for \( E_{\text{min}} \) and \( E_{\text{max}} \).

Each of the four value sets includes not only the finite nonzero values that are ascribed to it above, but also the five values positive zero, negative zero, positive infinity, negative infinity, and NaN.
Note that the constraints in Table 3.1 are designed so that every element of the float value set is necessarily also an element of the float-extended-exponent value set, the double value set, and the double-extended-exponent value set. Likewise, each element of the double value set is necessarily also an element of the double-extended-exponent value set. Each extended-exponent value set has a larger range of exponent values than the corresponding standard value set, but does not have more precision.

The elements of the float value set are exactly the values that can be represented using the single floating-point format defined in the IEEE 754 standard, except that there is only one NaN value (IEEE 754 specifies $2^{24} - 2$ distinct NaN values). The elements of the double value set are exactly the values that can be represented using the double floating-point format defined in the IEEE 754 standard, except that there is only one NaN value (IEEE 754 specifies $2^{53} - 2$ distinct NaN values). Note, however, that the elements of the float-extended-exponent and double-extended-exponent value sets defined here do not correspond to the values that be represented using IEEE 754 single extended and double extended formats, respectively. This specification does not mandate a specific representation for the values of the floating-point value sets except where floating-point values must be represented in the class file format (§4.4.4, §4.4.5).

The float, float-extended-exponent, double, and double-extended-exponent value sets are not types. It is always correct for an implementation of the Java virtual machine to use an element of the float value set to represent a value of type float; however, it may be permissible in certain contexts for an implementation to use an element of the float-extended-exponent value set instead. Similarly, it is always correct for an implementation to use an element of the double value set to represent a value of type double; however, it may be permissible in certain contexts for an implementation to use an element of the double-extended-exponent value set instead.

Except for NaNs, values of the floating-point value sets are ordered. When arranged from smallest to largest, they are negative infinity, negative finite values, positive and negative zero, positive finite values, and positive infinity.

Floating-point positive zero and floating-point negative zero compare as equal, but there are other operations that can distinguish them; for example, dividing 1.0 by 0.0 produces positive infinity, but dividing 1.0 by -0.0 produces negative infinity.

NaNs are unordered, so numerical comparisons and tests for numerical equality have the value false if either or both of their operands are NaN. In particular, a test for numerical equality of a value against itself has the value false if and
only if the value is NaN. A test for numerical inequality has the value true if either operand is NaN.

3.3.3 The returnAddress Type and Values

The returnAddress type is used by the Java virtual machine’s jsr, ret, and jsr_w instructions. The values of the returnAddress type are pointers to the opcodes of Java virtual machine instructions. Unlike the numeric primitive types, the returnAddress type does not correspond to any Java programming language type and cannot be modified by the running program.

3.3.4 The boolean Type

Although the Java virtual machine defines a boolean type, it only provides very limited support for it. There are no Java virtual machine instructions solely dedicated to operations on boolean values. Instead, expressions in the Java programming language that operate on boolean values are compiled to use values of the Java virtual machine int data type.

The Java virtual machine does directly support boolean arrays. Its newarray instruction enables creation of boolean arrays. Arrays of type boolean are accessed and modified using the byte array instructions baload and bastore.2

The Java virtual machine encodes boolean array components using 1 to represent true and 0 to represent false. Where Java programming language boolean values are mapped by compilers to values of Java virtual machine type int, the compilers must use the same encoding.

3.4 Reference Types and Values

There are three kinds of reference types: class types, array types, and interface types. Their values are references to dynamically created class instances, arrays, or class instances or arrays that implement interfaces, respectively. A reference value may also be the special null reference, a reference to no object, which will be denoted here by null. The null reference initially has no runtime type, but may be cast to any type (JLS3 §4.1).

2 In Sun’s Java virtual machine implementation, boolean arrays in the Java programming language are encoded as Java virtual machine byte arrays, using 8 bits per boolean element.
The Java virtual machine specification does not mandate a concrete value encoding null.

3.5 Runtime Data Areas

The Java virtual machine defines various runtime data areas that are used during execution of a program. Some of these data areas are created on Java virtual machine start-up and are destroyed only when the Java virtual machine exits. Other data areas are per thread. Per-thread data areas are created when a thread is created and destroyed when the thread exits.

3.5.1 The pc Register

The Java virtual machine can support many threads of execution at once (JLS3 §17). Each Java virtual machine thread has its own pc (program counter) register. At any point, each Java virtual machine thread is executing the code of a single method, namely the current method (§3.6) for that thread. If that method is not native, the pc register contains the address of the Java virtual machine instruction currently being executed. If the method currently being executed by the thread is native, the value of the Java virtual machine’s pc register is undefined. The Java virtual machine’s pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.

3.5.2 Java Virtual Machine Stacks

Each Java virtual machine thread has a private Java virtual machine stack, created at the same time as the thread.3 A Java virtual machine stack stores frames (§3.6). A Java virtual machine stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return. Because the Java virtual machine stack is never manipulated directly except to push and pop frames, frames may be heap allocated. The memory for a Java virtual machine stack does not need to be contiguous.

The Java virtual machine specification permits Java virtual machine stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the Java virtual machine stacks are of a fixed size, the size of each Java virtual machine stack may be chosen independently when that stack is

---

3 In the first edition of this specification, the Java virtual machine stack was known as the Java stack.
created. A Java virtual machine implementation may provide the programmer or the user control over the initial size of Java virtual machine stacks, as well as, in the case of dynamically expanding or contracting Java virtual machine stacks, control over the maximum and minimum sizes.

The following exceptional conditions are associated with Java virtual machine stacks:

- If the computation in a thread requires a larger Java virtual machine stack than is permitted, the Java virtual machine throws a `StackOverflowError`.

- If Java virtual machine stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to effect the expansion, or if insufficient memory can be made available to create the initial Java virtual machine stack for a new thread, the Java virtual machine throws an `OutOfMemoryError`.

### 3.5.3 Heap

The Java virtual machine has a *heap* that is shared among all Java virtual machine threads. The heap is the runtime data area from which memory for all class instances and arrays is allocated.

The heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system (known as a *garbage collector*); objects are never explicitly deallocated. The Java virtual machine assumes no particular type of automatic storage management system, and the storage management technique may be chosen according to the implementor’s system requirements. The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.

A Java virtual machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.

The following exceptional condition is associated with the heap:
• If a computation requires more heap than can be made available by the automatic storage management system, the Java virtual machine throws an OutOfMemoryError.

3.5.4 Method Area

The Java virtual machine has a method area that is shared among all Java virtual machine threads. The method area is analogous to the storage area for compiled code of a conventional language or analogous to the “text” segment in a UNIX® process. It stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods (§3.9) used in class and instance initialization and interface initialization.

The method area is created on virtual machine start-up. Although the method area is logically part of the heap, simple implementations may choose not to either garbage collect or compact it. This version of the Java virtual machine specification does not mandate the location of the method area or the policies used to manage compiled code. The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary. The memory for the method area does not need to be contiguous.

A Java virtual machine implementation may provide the programmer or the user control over the initial size of the method area, as well as, in the case of a varying-size method area, control over the maximum and minimum method area size.

The following exceptional condition is associated with the method area:

• If memory in the method area cannot be made available to satisfy an allocation request, the Java virtual machine throws an OutOfMemoryError.

3.5.5 Runtime Constant Pool

A runtime constant pool is a per-class or per-interface runtime representation of the constant_pool table in a class file (§4.4). It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at runtime. The runtime constant pool serves a function similar to that of a symbol table for a conventional programming language, although it contains a wider range of data than a typical symbol table.
Each runtime constant pool is allocated from the Java virtual machine’s method area (§3.5.4). The runtime constant pool for a class or interface is constructed when the class or interface is created (§5.3) by the Java virtual machine.

The following exceptional condition is associated with the construction of the runtime constant pool for a class or interface:

- When creating a class or interface, if the construction of the runtime constant pool requires more memory than can be made available in the method area of the Java virtual machine, the Java virtual machine throws an OutOfMemoryError.


3.5.6 Native Method Stacks

An implementation of the Java virtual machine may use conventional stacks, colloquially called “C stacks,” to support native methods, methods written in a language other than the Java programming language. Native method stacks may also be used by the implementation of an interpreter for the Java virtual machine’s instruction set in a language such as C. Java virtual machine implementations that cannot load native methods and that do not themselves rely on conventional stacks need not supply native method stacks. If supplied, native method stacks are typically allocated per thread when each thread is created.

The Java virtual machine specification permits native method stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the native method stacks are of a fixed size, the size of each native method stack may be chosen independently when that stack is created. In any case, a Java virtual machine implementation may provide the programmer or the user control over the initial size of the native method stacks. In the case of varying-size native method stacks, it may also make available control over the maximum and minimum method stack sizes.

The following exceptional conditions are associated with native method stacks:
• If the computation in a thread requires a larger native method stack than is permitted, the Java virtual machine throws a `StackOverflowError`.

• If native method stacks can be dynamically expanded and native method stack expansion is attempted but insufficient memory can be made available, or if insufficient memory can be made available to create the initial native method stack for a new thread, the Java virtual machine throws an `OutOfMemoryError`.

### 3.6 Frames

A frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception). Frames are allocated from the Java virtual machine stack (§3.5.2) of the thread creating the frame. Each frame has its own array of local variables (§3.6.1), its own operand stack (§3.6.2), and a reference to the runtime constant pool (§3.5.5) of the class of the current method.

The sizes of the local variable array and the operand stack are determined at compile time and are supplied along with the code for the method associated with the frame (§4.7.3). Thus the size of the frame data structure depends only on the implementation of the Java virtual machine, and the memory for these structures can be allocated simultaneously on method invocation.

Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the current frame, and its method is known as the current method. The class in which the current method is defined is the current class. Operations on local variables and the operand stack are typically with reference to the current frame.

A frame ceases to be current if its method invokes another method or if its method completes. When a method is invoked, a new frame is created and becomes current when control transfers to the new method. On method return, the current frame passes back the result of its method invocation, if any, to the previous frame. The current frame is then discarded as the previous frame becomes the current one.

Note that a frame created by a thread is local to that thread and cannot be referenced by any other thread.
3.6.1 Local Variables

Each frame (§3.6) contains an array of variables known as its local variables. The length of the local variable array of a frame is determined at compile time and supplied in the binary representation of a class or interface along with the code for the method associated with the frame (§4.7.3).

A single local variable can hold a value of type boolean, byte, char, short, int, float, reference, or returnAddress. A pair of local variables can hold a value of type long or double.

Local variables are addressed by indexing. The index of the first local variable is zero. An integer is considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array.

A value of type long or type double occupies two consecutive local variables. Such a value may only be addressed using the lesser index. For example, a value of type double stored in the local variable array at index n actually occupies the local variables with indices n and n + 1; however, the local variable at index n + 1 cannot be loaded from. It can be stored into. However, doing so invalidates the contents of local variable n.

The Java virtual machine does not require n to be even. In intuitive terms, values of types double and long need not be 64-bit aligned in the local variables array. Implementors are free to decide the appropriate way to represent such values using the two local variables reserved for the value.

The Java virtual machine uses local variables to pass parameters on method invocation. On class method invocation any parameters are passed in consecutive local variables starting from local variable 0. On instance method invocation, local variable 0 is always used to pass a reference to the object on which the instance method is being invoked (this in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable 1.

3.6.2 Operand Stacks

Each frame (§3.6) contains a last-in-first-out (LIFO) stack known as its operand stack. The maximum depth of the operand stack of a frame is determined at compile time and is supplied along with the code for the method associated with the frame (§4.7.3).

Where it is clear by context, we will sometimes refer to the operand stack of the current frame as simply the operand stack.
The operand stack is empty when the frame that contains it is created. The Java virtual machine supplies instructions to load constants or values from local variables or fields onto the operand stack. Other Java virtual machine instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack. The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

For example, the \texttt{iadd} instruction adds two \texttt{int} values together. It requires that the \texttt{int} values to be added be the top two values of the operand stack, pushed there by previous instructions. Both of the \texttt{int} values are popped from the operand stack. They are added, and their sum is pushed back onto the operand stack. Subcomputations may be nested on the operand stack, resulting in values that can be used by the encompassing computation.

Each entry on the operand stack can hold a value of any Java virtual machine type, including a value of type \texttt{long} or type \texttt{double}.

Values from the operand stack must be operated upon in ways appropriate to their types. It is not possible, for example, to push two \texttt{int} values and subsequently treat them as a \texttt{long} or to push two \texttt{float} values and subsequently add them with an \texttt{iadd} instruction. A small number of Java virtual machine instructions (the \texttt{dup} instructions and \texttt{swap}) operate on runtime data areas as raw values without regard to their specific types; these instructions are defined in such a way that they cannot be used to modify or break up individual values. These restrictions on operand stack manipulation are enforced through \texttt{class} file verification (§4.10).

At any point in time an operand stack has an associated depth, where a value of type \texttt{long} or \texttt{double} contributes two units to the depth and a value of any other type contributes one unit.

### 3.6.3 Dynamic Linking

Each frame (§3.6) contains a reference to the runtime constant pool (§3.5.5) for the type of the current method to support dynamic linking of the method code. The \texttt{class} file code for a method refers to methods to be invoked and variables to be accessed via symbolic references. Dynamic linking translates these symbolic method references into concrete method references, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.
This late binding of the methods and variables makes changes in other classes that a method uses less likely to break this code.

3.6.4 Normal Method Invocation Completion

A method invocation completes normally if that invocation does not cause an exception (§3.10, JLS3 §9) to be thrown, either directly from the Java virtual machine or as a result of executing an explicit throw statement. If the invocation of the current method completes normally, then a value may be returned to the invoking method. This occurs when the invoked method executes one of the return instructions (§3.11.8), the choice of which must be appropriate for the type of the value being returned (if any).

The current frame (§3.6) is used in this case to restore the state of the invoker, including its local variables and operand stack, with the program counter of the invoker appropriately incremented to skip past the method invocation instruction. Execution then continues normally in the invoking method’s frame with the returned value (if any) pushed onto the operand stack of that frame.

3.6.5 Abrupt Method Invocation Completion

A method invocation completes abruptly if execution of a Java virtual machine instruction within the method causes the Java virtual machine to throw an exception (§3.10, JLS3 §9), and that exception is not handled within the method. Execution of an athrow instruction also causes an exception to be explicitly thrown and, if the exception is not caught by the current method, results in abrupt method invocation completion. A method invocation that completes abruptly never returns a value to its invoker.

3.6.6 Additional Information

A frame may be extended with additional implementation-specific information, such as debugging information.

3.7 Representation of Objects

The Java virtual machine does not mandate any particular internal structure for objects.⁴
3.8 Floating-Point Arithmetic


3.8.1 Java Virtual Machine Floating-Point Arithmetic and IEEE 754

The key differences between the floating-point arithmetic supported by the Java virtual machine and the IEEE 754 standard are:

- The floating-point operations of the Java virtual machine do not throw exceptions, trap, or otherwise signal the IEEE 754 exceptional conditions of invalid operation, division by zero, overflow, underflow, or inexact. The Java virtual machine has no signaling NaN value.

- The Java virtual machine does not support IEEE 754 signaling floating-point comparisons.

- The rounding operations of the Java virtual machine always use IEEE 754 round to nearest mode. Inexact results are rounded to the nearest representable value, with ties going to the value with a zero least-significant bit. This is the IEEE 754 default mode. But Java virtual machine instructions that convert values of floating-point types to values of integral types round toward zero. The Java virtual machine does not give any means to change the floating-point rounding mode.

- The Java virtual machine does not support either the IEEE 754 single extended or double extended format, except insofar as the double and double-extended-exponent value sets may be said to support the single extended format. The float-extended-exponent and double-extended-exponent value sets, which may optionally be supported, do not correspond to the values of the IEEE 754 extended formats: the IEEE 754 extended formats require extended precision as well as extended exponent range.

---

4 In some of Sun’s implementations of the Java virtual machine, a reference to a class instance is a pointer to a *handle* that is itself a pair of pointers: one to a table containing the methods of the object and a pointer to the *Class* object that represents the type of the object, and the other to the memory allocated from the heap for the object data.
3.8.2 Floating-Point Modes

Every method has a floating-point mode, which is either FP-strict or not FP-strict. The floating-point mode of a method is determined by the setting of the ACC_STRICT flag of the access_flags item of the method_info structure (§4.6) defining the method. A method for which this flag is set is FP-strict; otherwise, the method is not FP-strict.

Note that this mapping of the ACC_STRICT flag implies that methods in classes compiled by a compiler in JDK release 1.1 or earlier are effectively not FP-strict.

We will refer to an operand stack as having a given floating-point mode when the method whose invocation created the frame containing the operand stack has that floating-point mode. Similarly, we will refer to a Java virtual machine instruction as having a given floating-point mode when the method containing that instruction has that floating-point mode.

If a float-extended-exponent value set is supported (§3.3.2), values of type float on an operand stack that is not FP-strict may range over that value set except where prohibited by value set conversion (§3.8.3). If a double-extended-exponent value set is supported (§3.3.2), values of type double on an operand stack that is not FP-strict may range over that value set except where prohibited by value set conversion.

In all other contexts, whether on the operand stack or elsewhere, and regardless of floating-point mode, floating-point values of type float and double may only range over the float value set and double value set, respectively. In particular, class and instance fields, array elements, local variables, and method parameters may only contain values drawn from the standard value sets.

3.8.3 Value Set Conversion

An implementation of the Java virtual machine that supports an extended floating-point value set is permitted or required, under specified circumstances, to map a value of the associated floating-point type between the extended and the standard value sets. Such a value set conversion is not a type conversion, but a mapping between the value sets associated with the same type.

Where value set conversion is indicated, an implementation is permitted to perform one of the following operations on a value:
• If the value is of type `float` and is not an element of the float value set, it maps the value to the nearest element of the float value set.

• If the value is of type `double` and is not an element of the double value set, it maps the value to the nearest element of the double value set.

In addition, where value set conversion is indicated certain operations are required:

• Suppose execution of a Java virtual machine instruction that is not FP-strict causes a value of type `float` to be pushed onto an operand stack that is FP-strict, passed as a parameter, or stored into a local variable, a field, or an element of an array. If the value is not an element of the float value set, it maps the value to the nearest element of the float value set.

• Suppose execution of a Java virtual machine instruction that is not FP-strict causes a value of type `double` to be pushed onto an operand stack that is FP-strict, passed as a parameter, or stored into a local variable, a field, or an element of an array. If the value is not an element of the double value set, it maps the value to the nearest element of the double value set.

Such required value set conversions may occur as a result of passing a parameter of a floating-point type during method invocation, including native method invocation; returning a value of a floating-point type from a method that is not FP-strict to a method that is FP-strict; or storing a value of a floating-point type into a local variable, a field, or an array in a method that is not FP-strict.

Not all values from an extended-exponent value set can be mapped exactly to a value in the corresponding standard value set. If a value being mapped is too large to be represented exactly (its exponent is greater than that permitted by the standard value set), it is converted to a (positive or negative) infinity of the corresponding type. If a value being mapped is too small to be represented exactly (its exponent is smaller than that permitted by the standard value set), it is rounded to the nearest of a representable denormalized value or zero of the same sign.

Value set conversion preserves infinities and NaNs and cannot change the sign of the value being converted. Value set conversion has no effect on a value that is not of a floating-point type.
3.9 Specially Named Initialization Methods

At the level of the Java virtual machine, every constructor (JLS3 §8.8) appears as an instance initialization method that has the special name `<init>`. This name is supplied by a compiler. Because the name `<init>` is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Instance initialization methods may be invoked only within the Java virtual machine by the `invokespecial` instruction, and they may be invoked only on uninitialized class instances. An instance initialization method takes on the access permissions (JLS3 §6.6) of the constructor from which it was derived.

A class or interface has at most one class or interface initialization method and is initialized (§5.5) by invoking that method. The initialization method of a class or interface is static, takes no arguments, and has the special name `<clinit>`. This name is supplied by a compiler. Because the name `<clinit>` is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Class and interface initialization methods are invoked implicitly by the Java virtual machine; they are never invoked directly from any Java virtual machine instruction, but are invoked only indirectly as part of the class initialization process.

3.10 Exceptions

An exception in the Java virtual machine is represented by an instance of the class `Throwable` or one of its subclasses. Throwing an exception results in an immediate nonlocal transfer of control from the point where the exception was thrown.

Most exceptions occur synchronously as a result of an action by the thread in which they occur. An asynchronous exception, by contrast, can potentially occur at any point in the execution of a program. The Java virtual machine throws an exception for one of three reasons:

1. An abnormal execution condition was synchronously detected by the Java virtual machine. These exceptions are not thrown at an arbitrary point in the program, but only synchronously after execution of an instruction that either:
   - Specifies the exception as a possible result, such as:

---

5 Other methods named `<clinit>` in a class file are of no consequence. They are not class or interface initialization methods. They cannot be invoked by any Java virtual machine instruction and are never invoked by the Java virtual machine itself.
• When the instruction embodies an operation that violates the semantics of the Java programming language, for example indexing outside the bounds of an array.

• When an error occurs in loading or linking part of the program.
  • Causes some limit on a resource to be exceeded, for example when too much memory is used.

2. An `athrow` instruction was executed.

3. An asynchronous exception occurred because:
  • The `stop` method of class `Thread` or `ThreadGroup` was invoked, or
  • An internal error occurred in the Java virtual machine implementation.

The `stop` methods may be invoked by one thread to affect another thread or all the threads in a specified thread group. They are asynchronous because they may occur at any point in the execution of the other thread or threads. An internal error is considered asynchronous (§6.3).

A Java virtual machine may permit a small but bounded amount of execution to occur before an asynchronous exception is thrown. This delay is permitted to allow optimized code to detect and throw these exceptions at points where it is practical to handle them while obeying the semantics of the Java programming language.\(^6\)

Exceptions thrown by the Java virtual machine are precise: when the transfer of control takes place, all effects of the instructions executed before the point from which the exception is thrown must appear to have taken place. No instructions that occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the instructions which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program.

Each method in the Java virtual machine may be associated with zero or more exception handlers. An exception handler specifies the range of offsets into the Java virtual machine code implementing the method for which the exception han-

---

\(^6\) A simple implementation might poll for asynchronous exceptions at the point of each control transfer instruction. Since a program has a finite size, this provides a bound on the total delay in detecting an asynchronous exception. Since no asynchronous exception will occur between control transfers, the code generator has some flexibility to reorder computation between control transfers for greater performance. The paper *Polling Efficiently on Stock Hardware* by Marc Feeley, *Proc. 1993 Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, pp. 179–187, is recommended as further reading.
dler is active, describes the type of exception that the exception handler is able to handle, and specifies the location of the code that is to handle that exception. An exception matches an exception handler if the offset of the instruction that caused the exception is in the range of offsets of the exception handler and the exception type is the same class as or a subclass of the class of exception that the exception handler handles. When an exception is thrown, the Java virtual machine searches for a matching exception handler in the current method. If a matching exception handler is found, the system branches to the exception handling code specified by the matched handler.

If no such exception handler is found in the current method, the current method invocation completes abruptly (§3.6.5). On abrupt completion, the operand stack and local variables of the current method invocation are discarded, and its frame is popped, reinstating the frame of the invoking method. The exception is then rethrown in the context of the invoker’s frame and so on, continuing up the method invocation chain. If no suitable exception handler is found before the top of the method invocation chain is reached, the execution of the thread in which the exception was thrown is terminated.

The order in which the exception handlers of a method are searched for a match is important. Within a class file the exception handlers for each method are stored in a table (§4.7.3). At runtime, when an exception is thrown, the Java virtual machine searches the exception handlers of the current method in the order that they appear in the corresponding exception handler table in the class file, starting from the beginning of that table.

Note that the Java virtual machine does not enforce nesting of or any ordering of the exception table entries of a method. The exception handling semantics of the Java programming language are implemented only through cooperation with the compiler (§7.12). When class files are generated by some other means, the defined search procedure ensures that all Java virtual machines will behave consistently.

### 3.11 Instruction Set Summary

A Java virtual machine instruction consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation. Many instructions have no operands and consist only of an opcode.

Ignoring exceptions, the inner loop of a Java virtual machine interpreter is effectively
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);

The number and size of the operands are determined by the opcode. If an operand is more than one byte in size, then it is stored in big-endian order—high-order byte first. For example, an unsigned 16-bit index into the local variables is stored as two unsigned bytes, \texttt{byte1} and \texttt{byte2}, such that its value is

\[(\texttt{byte1} \ll 8) | \texttt{byte2}\]

The bytecode instruction stream is only single-byte aligned. The two exceptions are the \texttt{tableswitch} and \texttt{lookupswitch} instructions, which are padded to force internal alignment of some of their operands on 4-byte boundaries.

The decision to limit the Java virtual machine opcode to a byte and to forgo data alignment within compiled code reflects a conscious bias in favor of compactness, possibly at the cost of some performance in naive implementations. A one-byte opcode also limits the size of the instruction set. Not assuming data alignment means that immediate data larger than a byte must be constructed from bytes at runtime on many machines.

### 3.11.1 Types and the Java Virtual Machine

Most of the instructions in the Java virtual machine instruction set encode type information about the operations they perform. For instance, the \texttt{iload} instruction loads the contents of a local variable, which must be an \texttt{int}, onto the operand stack. The \texttt{fload} instruction does the same with a \texttt{float} value. The two instructions may have identical implementations, but have distinct opcodes.

For the majority of typed instructions, the instruction type is represented explicitly in the opcode mnemonic by a letter: \texttt{i} for an \texttt{int} operation, \texttt{l} for \texttt{long}, \texttt{s} for \texttt{short}, \texttt{b} for \texttt{byte}, \texttt{c} for \texttt{char}, \texttt{f} for \texttt{float}, \texttt{d} for \texttt{double}, and \texttt{a} for \texttt{reference}. Some instructions for which the type is unambiguous do not have a type letter in their mnemonic. For instance, \texttt{arraylength} always operates on an object that is an array. Some instructions, such as \texttt{goto}, an unconditional control transfer, do not operate on typed operands.

Given the Java virtual machine’s one-byte opcode size, encoding types into opcodes places pressure on the design of its instruction set. If each typed instruc-
tion supported all of the Java virtual machine’s runtime data types, there would be more instructions than could be represented in a byte. Instead, the instruction set of the Java virtual machine provides a reduced level of type support for certain operations. In other words, the instruction set is intentionally not orthogonal. Separate instructions can be used to convert between unsupported and supported data types as necessary.

Table 3.2 summarizes the type support in the instruction set of the Java virtual machine. A specific instruction, with type information, is built by replacing the T in the instruction template in the opcode column by the letter in the type column. If the type column for some instruction template and type is blank, then no instruction exists supporting that type of operation. For instance, there is a load instruction for type int, iload, but there is no load instruction for type byte.

Note that most instructions in Table 3.2 do not have forms for the integral types byte, char, and short. None have forms for the boolean type. Compilers encode loads of literal values of types byte and short using Java virtual machine instructions that sign-extend those values to values of type int at compile time or runtime. Loads of literal values of types boolean and char are encoded using instructions that zero-extend the literal to a value of type int at compile time or runtime. Likewise, loads from arrays of values of type boolean, byte, short, and char are encoded using Java virtual machine instructions that sign-extend or zero-extend the values to values of type int. Thus, most operations on values of actual types boolean, byte, char, and short are correctly performed by instructions operating on values of computational type int.
### Table 3.2  Type support in the Java virtual machine instruction set

<table>
<thead>
<tr>
<th>opcode</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
<th>char</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tpush</td>
<td>bipush</td>
<td>sipush</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tconst</td>
<td>iconst</td>
<td>ICONST</td>
<td>lconst</td>
<td>fconst</td>
<td>dconst</td>
<td>aconst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tload</td>
<td>iload</td>
<td>lload</td>
<td>float</td>
<td>dload</td>
<td>aload</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tstore</td>
<td>istore</td>
<td>lstore</td>
<td>fstore</td>
<td>dstore</td>
<td>astore</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tinc</td>
<td>iinc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taload</td>
<td>baload</td>
<td>saload</td>
<td>iload</td>
<td>lload</td>
<td>fload</td>
<td>dload</td>
<td>caload</td>
<td>aaload</td>
</tr>
<tr>
<td>Tastore</td>
<td>bastore</td>
<td>sastore</td>
<td>iastore</td>
<td>lastore</td>
<td>fastore</td>
<td>dastore</td>
<td>castore</td>
<td>aastore</td>
</tr>
<tr>
<td>Tadd</td>
<td>iadd</td>
<td>ladd</td>
<td>fadd</td>
<td>dadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tsub</td>
<td>isub</td>
<td>lsub</td>
<td>fsub</td>
<td>dsub</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmul</td>
<td>imul</td>
<td>lmul</td>
<td>fmul</td>
<td>dmul</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tdiv</td>
<td>idiv</td>
<td>ldiv</td>
<td>fdiv</td>
<td>ddiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trem</td>
<td>irem</td>
<td>lrem</td>
<td>frem</td>
<td>drem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tneg</td>
<td>ineg</td>
<td>lneg</td>
<td>fneg</td>
<td>dneg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshl</td>
<td>ishl</td>
<td>lshr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshr</td>
<td>ishr</td>
<td>lshr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tushr</td>
<td>iushr</td>
<td>lushr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tand</td>
<td>iand</td>
<td>land</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tor</td>
<td>ior</td>
<td>lor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Txor</td>
<td>ixor</td>
<td>lxor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td>i2b</td>
<td>i2s</td>
<td>i2l</td>
<td>i2f</td>
<td>i2d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l2T</td>
<td>l2i</td>
<td>l2f</td>
<td>l2d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2T</td>
<td>f2i</td>
<td>f2l</td>
<td>f2d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2T</td>
<td>d2i</td>
<td>d2l</td>
<td>d2f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td>lcmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmpf</td>
<td>fcmpl</td>
<td>dcmpl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmpg</td>
<td>fcmpg</td>
<td>dcmpg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if_TcmpOP</td>
<td>if_icmpOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treturn</td>
<td>ireturn</td>
<td>lreturn</td>
<td>freturn</td>
<td>dreturn</td>
<td>areturn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The mapping between Java virtual machine actual types and Java virtual machine computational types is summarized by Table 3.3.

Table 3.3  Java virtual machine actual and computational types

<table>
<thead>
<tr>
<th>Actual Type</th>
<th>Computational Type</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>byte</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>char</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>category 1</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>category 1</td>
</tr>
<tr>
<td>reference</td>
<td>reference</td>
<td>category 1</td>
</tr>
<tr>
<td>returnAddress</td>
<td>returnAddress</td>
<td>category 1</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>category 2</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>category 2</td>
</tr>
</tbody>
</table>

Certain Java virtual machine instructions such as *pop* and *swap* operate on the operand stack without regard to type; however, such instructions are constrained to use only on values of certain categories of computational types, also given in Table 3.3.

3.11.2  Load and Store Instructions

The load and store instructions transfer values between the local variables (§3.6.1) and the operand stack (§3.6.2) of a Java virtual machine frame (§3.6):

- Load a local variable onto the operand stack: `iload, iload_<n>, lload, lload_<n>, fload, fload_<n>, dload, dload_<n>, aload, aload_<n>`.  
- Store a value from the operand stack into a local variable: `istore, istore_<n>, lstore, lstore_<n>, fstore, fstore_<n>, dstore, dstore_<n>, astore, astore_<n>`.  
- Load a constant onto the operand stack: `bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, iconst_<i>, lconst_<l>, fconst_<f>, dconst_<d>`.  
- Gain access to more local variables using a wider index, or to a larger immediate operand: `wide`.  

Instructions that access fields of objects and elements of arrays (§3.11.5) also transfer data to and from the operand stack.

Instruction mnemonics shown above with trailing letters between angle brackets (for instance, *iload_<n>*)) denote families of instructions (with members *iload_0*, *iload_1*, *iload_2*, and *iload_3* in the case of *iload_<n>*). Such families of instructions are specializations of an additional generic instruction (*iload*) that takes one operand. For the specialized instructions, the operand is implicit and does not need to be stored or fetched. The semantics are otherwise the same (*iload_0* means the same thing as *iload* with the operand 0). The letter between the angle brackets specifies the type of the implicit operand for that family of instructions: for *<n>*., a nonnegative integer; for *<i>*., an *int*; for *<l>*., a *long*; for *<f>*., a *float*; and for *<d>*., a *double*. Forms for type *int* are used in many cases to perform operations on values of type *byte*, *char*, and *short* (§3.11.1).

This notation for instruction families is used throughout *The Java™ Virtual Machine Specification*.

### 3.11.3 Arithmetic Instructions

The arithmetic instructions compute a result that is typically a function of two values on the operand stack, pushing the result back on the operand stack. There are two main kinds of arithmetic instructions: those operating on integer values and those operating on floating-point values. Within each of these kinds, the arithmetic instructions are specialized to Java virtual machine numeric types. There is no direct support for integer arithmetic on values of the *byte*, *short*, and *char* types (§3.11.1), or for values of the *boolean* type; those operations are handled by instructions operating on type *int*. Integer and floating-point instructions also differ in their behavior on overflow and divide-by-zero. The arithmetic instructions are as follows:

- **Add:** *iadd*, *ladd*, *fadd*, *dadd*.
- **Subtract:** *isub*, *lsub*, *fsub*, *dsub*.
- **Multiply:** *imul*, *lmul*, *fmul*, *dmul*.
- **Divide:** *idiv*, *ldiv*, *fdiv*, *ddiv*.
- **Remainder:** *irem*, *lrem*, *frem*, *drem*.
- **Negate:** *ineg*, *lneg*, *fneg*, *dneg*.
- **Shift:** *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
• Bitwise OR: ior, lor.
• Bitwise AND: iand, land.
• Bitwise exclusive OR: ixor, lxor.
• Local variable increment: iinc.
• Comparison: dcmpg, dcml, fcmpg, fcmpl, lcmp.

The semantics of the Java programming language operators on integer and floating-point values (JLS3 §4.2.2, JLS3 §4.2.4) are directly supported by the semantics of the Java virtual machine instruction set.

The Java virtual machine does not indicate overflow during operations on integer data types. The only integer operations that can throw an exception are the integer divide instructions (idiv and ldiv) and the integer remainder instructions (irem and lrem), which throw an ArithmeticException if the divisor is zero.

Java virtual machine operations on floating-point numbers behave as specified in IEEE 754. In particular, the Java virtual machine requires full support of IEEE 754 denormalized floating-point numbers and gradual underflow, which make it easier to prove desirable properties of particular numerical algorithms.

The Java virtual machine requires that floating-point arithmetic behave as if every floating-point operator rounded its floating-point result to the result precision. Inexact results must be rounded to the representable value nearest to the infinitely precise result; if the two nearest representable values are equally near, the one having a least significant bit of zero is chosen. This is the IEEE 754 standard’s default rounding mode, known as round to nearest mode.

The Java virtual machine uses the IEEE 754 round towards zero mode when converting a floating-point value to an integer. This results in the number being truncated; any bits of the significand that represent the fractional part of the operand value are discarded. Round towards zero mode chooses as its result the type’s value closest to, but no greater in magnitude than, the infinitely precise result.

The Java virtual machine’s floating-point operators do not throw runtime exceptions (not to be confused with IEEE 754 floating-point exceptions). An operation that overflows produces a signed infinity, an operation that underflows produces a denormalized value or a signed zero, and an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result.
Comparisons on values of type long (lcmp) perform a signed comparison. Comparisons on values of floating-point types (dcmpg, dcmpl, fcmpg, fcmpl) are performed using IEEE 754 nonsignaling comparisons.

### 3.11.4 Type Conversion Instructions

The type conversion instructions allow conversion between Java virtual machine numeric types. These may be used to implement explicit conversions in user code or to mitigate the lack of orthogonality in the instruction set of the Java virtual machine.

The Java virtual machine directly supports the following widening numeric conversions:

- int to long, float, or double
- long to float or double
- float to double

The mnemonics for these opcodes are straightforward given the naming conventions for typed instructions and the punning use of 2 to mean “to.” For instance, the i2d instruction converts an int value to a double. Widening numeric conversions do not lose information about the overall magnitude of a numeric value. Indeed, conversions widening from int to long and int to double do not lose any information at all; the numeric value is preserved exactly. Conversions widening from float to double that are FP-strict (§3.8.2) also preserve the numeric value exactly; however, such conversions that are not FP-strict may lose information about the overall magnitude of the converted value.

Conversion of an int or a long value to float, or of a long value to double, may lose precision, that is, may lose some of the least significant bits of the value; the resulting floating-point value is a correctly rounded version of the integer value, using IEEE 754 round to nearest mode.

A widening numeric conversion of an int to a long simply sign-extends the two’s-complement representation of the int value to fill the wider format. A widening numeric conversion of a char to an integral type zero-extends the representation of the char value to fill the wider format.

Despite the fact that loss of precision may occur, widening numeric conversions never cause the Java virtual machine to throw a runtime exception (not to be confused with an IEEE 754 floating-point exception).
Note that widening numeric conversions do not exist from integral types byte, char, and short to type int. As noted in §3.11.1, values of type byte, char, and short are internally widened to type int, making these conversions implicit.

The Java virtual machine also directly supports the following narrowing numeric conversions:

- int to byte, short, or char
- long to int
- float to int or long
- double to int, long, or float

The narrowing numeric conversion instructions are i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, and d2f. A narrowing numeric conversion can result in a value of different sign, a different order of magnitude, or both; it may thereby lose precision.

A narrowing numeric conversion of an int or long to an integral type T simply discards all but the N lowest-order bits, where N is the number of bits used to represent type T. This may cause the resulting value not to have the same sign as the input value.

In a narrowing numeric conversion of a floating-point value to an integral type T, where T is either int or long, the floating-point value is converted as follows:

- If the floating-point value is NaN, the result of the conversion is an int or long 0.
- Otherwise, if the floating-point value is not an infinity, the floating-point value is rounded to an integer value V using IEEE 754 round towards zero mode. There are two cases:
  - If T is long and this integer value can be represented as a long, then the result is the long value V.
  - If T is of type int and this integer value can be represented as an int, then the result is the int value V.
- Otherwise:
  - Either the value must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type int or long.
• Or the value must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type int or long.

A narrowing numeric conversion from double to float behaves in accordance with IEEE 754. The result is correctly rounded using IEEE 754 round to nearest mode. A value too small to be represented as a float is converted to a positive or negative zero of type float; a value too large to be represented as a float is converted to a positive or negative infinity. A double NaN is always converted to a float NaN.

Despite the fact that overflow, underflow, or loss of precision may occur, narrowing conversions among numeric types never cause the Java virtual machine to throw a runtime exception (not to be confused with an IEEE 754 floating-point exception).

3.11.5 Object Creation and Manipulation

Although both class instances and arrays are objects, the Java virtual machine creates and manipulates class instances and arrays using distinct sets of instructions:

• Create a new class instance: new.

• Create a new array: newarray, anewarray, multianewarray.

• Access fields of classes (static fields, known as class variables) and fields of class instances (non-static fields, known as instance variables): getfield, putfield, getstatic, putstatic.

• Load an array component onto the operand stack: baload, caload, saload, iaload, laload, faload, daload, aaload.

• Store a value from the operand stack as an array component: bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore.

• Get the length of array: arraylength.

• Check properties of class instances or arrays: instanceof, checkcast.

3.11.6 Operand Stack Management Instructions

A number of instructions are provided for the direct manipulation of the operand stack: pop, pop2, dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2, swap.
3.11.7 Control Transfer Instructions

The control transfer instructions conditionally or unconditionally cause the Java virtual machine to continue execution with an instruction other than the one following the control transfer instruction. They are:

- Conditional branch: `ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpeq`, `if_acmpne`.
- Compound conditional branch: `tableswitch`, `lookupswitch`.
- Unconditional branch: `goto`, `goto_w`, `jsr`, `jsr_w`, `ret`.

The Java virtual machine has distinct sets of instructions that conditionally branch on comparison with data of `int` and reference types. It also has distinct conditional branch instructions that test for the null reference and thus is not required to specify a concrete value for `null` (§3.4).

Conditional branches on comparisons between data of types `boolean`, `byte`, `char`, and `short` are performed using `int` comparison instructions (§3.11.1). A conditional branch on a comparison between data of types `long`, `float`, or `double` is initiated using an instruction that compares the data and produces an `int` result of the comparison (§3.11.3). A subsequent `int` comparison instruction tests this result and effects the conditional branch. Because of its emphasis on `int` comparisons, the Java virtual machine provides a rich complement of conditional branch instructions for type `int`.

All `int` conditional control transfer instructions perform signed comparisons.

3.11.8 Method Invocation and Return Instructions

The following four instructions invoke methods:

- `invokevirtual` invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the normal method dispatch in the Java programming language.
- `invokeinterface` invokes an interface method, searching the methods implemented by the particular runtime object to find the appropriate method.
- `invokespecial` invokes an instance method requiring special handling, whether an instance initialization method (§3.9), a `private` method, or a superclass method.
• **invokes\text{static}** invokes a class (static) method in a named class.

The method return instructions, which are distinguished by return type, are \textit{ireturn} (used to return values of type \texttt{boolean}, \texttt{byte}, \texttt{char}, \texttt{short}, or \texttt{int}), \textit{lreturn}, \textit{freturn}, \textit{dreturn}, and \textit{areturn}. In addition, the \textit{return} instruction is used to return from methods declared to be \texttt{void}, instance initialization methods, and class or interface initialization methods.

### 3.11.9 Throwing Exceptions

An exception is thrown programmatically using the \textit{athrow} instruction. Exceptions can also be thrown by various Java virtual machine instructions if they detect an abnormal condition.

### 3.11.10 Synchronization

The Java virtual machine supports synchronization of both methods and sequences of instructions within a method by a single synchronization construct: the \textit{monitor}.

Method-level synchronization is performed implicitly, as part of method invocation and return (§3.11.8). A synchronized method is distinguished in the runtime constant pool’s \texttt{method\\_info} structure (§4.6) by the \texttt{ACC\\_SYNCHRONIZED} flag, which is checked by the method invocation instructions. When invoking a method for which \texttt{ACC\\_SYNCHRONIZED} is set, the executing thread enters a monitor, invokes the method itself, and exits the monitor whether the method invocation completes normally or abruptly. During the time the executing thread owns the monitor, no other thread may enter it. If an exception is thrown during invocation of the synchronized method and the synchronized method does not handle the exception, the monitor for the method is automatically exited before the exception is rethrown out of the synchronized method.

Synchronization of sequences of instructions is typically used to encode the synchronized block of the Java programming language. The Java virtual machine supplies the \texttt{monitorenter} and \texttt{monitorexit} instructions to support such language constructs. Proper implementation of synchronized blocks requires cooperation from a compiler targeting the Java virtual machine (see Section 7.15, “Synchronization”).

\textit{Structured locking} is the situation when, during a method invocation, every exit on a given monitor matches a preceding entry on that monitor. Since there is no assurance that all code submitted to the Java virtual machine will perform
structured locking, implementations of the Java virtual machine are permitted but not required to enforce both of the following two rules guaranteeing structured locking. Let $T$ be a thread and $M$ be a monitor. Then:

1. The number of monitor entries performed by $T$ on $M$ during a method invocation must equal the number of monitor exits performed by $T$ on $M$ during the method invocation whether the method invocation completes normally or abruptly.

2. At no point during a method invocation may the number of monitor exits performed by $T$ on $M$ since the method invocation exceed the number of monitor entries performed by $T$ on $M$ since the method invocation.

Note that the monitor entry and exit automatically performed by the Java virtual machine when invoking a synchronized method are considered to occur during the calling method’s invocation.

### 3.12 Class Libraries

The Java virtual machine must provide sufficient support for the implementation of the class libraries of the associated platform. Some of the classes in these libraries cannot be implemented without the cooperation of the Java virtual machine.

Classes that might require special support from the Java virtual machine include those that support:

- Reflection, such as the classes in the package `java.lang.reflect` and the class `Class`.
- Loading and creation of a class or interface. The most obvious example is the class `ClassLoader`.
- Linking and initialization of a class or interface. The example classes cited above fall into this category as well.
- Security, such as the classes in the package `java.security` and other classes such as `SecurityManager`.
- Multithreading, such as the class `Thread`.
- Weak references, such as the classes in the package `java.lang.ref`. 
The list above is meant to be illustrative rather than comprehensive. An exhaustive list of these classes or of the functionality they provide is beyond the scope of this book. See the specifications of the Java platform class libraries for details.

### 3.13 Public Design, Private Implementation

Thus far this book has sketched the public view of the Java virtual machine: the class file format and the instruction set. These components are vital to the hardware-, operating system-, and implementation-independence of the Java virtual machine. The implementor may prefer to think of them as a means to securely communicate fragments of programs between hosts each implementing the Java platform, rather than as a blueprint to be followed exactly.

It is important to understand where the line between the public design and the private implementation lies. A Java virtual machine implementation must be able to read class files and must exactly implement the semantics of the Java virtual machine code therein. One way of doing this is to take this document as a specification and to implement that specification literally. But it is also perfectly feasible and desirable for the implementor to modify or optimize the implementation within the constraints of this specification. So long as the class file format can be read and the semantics of its code are maintained, the implementor may implement these semantics in any way. What is “under the hood” is the implementor’s business, as long as the correct external interface is carefully maintained.\(^7\)

The implementor can use this flexibility to tailor Java virtual machine implementations for high performance, low memory use, or portability. What makes sense in a given implementation depends on the goals of that implementation. The range of implementation options includes the following:

- Translating Java virtual machine code at load time or during execution into the instruction set of another virtual machine.

\(^7\) There are some exceptions: debuggers, profilers, and just-in-time code generators can each require access to elements of the Java virtual machine that are normally considered to be “under the hood.” Where appropriate, Sun works with other Java virtual machine implementors and tools vendors to develop common interfaces to the Java virtual machine for use by such tools, and to promote those interfaces across the industry. Information on publicly available low-level interfaces to the Java virtual machine will be made available at [http://java.sun.com](http://java.sun.com).
• Translating Java virtual machine code at load time or during execution into the native instruction set of the host CPU (sometimes referred to as *just-in-time*, or *JIT*, code generation).

The existence of a precisely defined virtual machine and object file format need not significantly restrict the creativity of the implementor. The Java virtual machine is designed to support many different implementations, providing new and interesting solutions while retaining compatibility between implementations.
CHAPTER 4

The class File Format

This chapter describes the Java virtual machine class file format. Each class file contains the definition of a single class or interface. Although a class or interface need not have an external representation literally contained in a file (for instance, because the class is generated by a class loader), we will colloquially refer to any valid representation of a class or interface as being in the class file format.

A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively. Multibyte data items are always stored in big-endian order, where the high bytes come first. In the Java platform, this format is supported by interfaces java.io.DataInput and java.io.DataOutput and classes such as java.io.DataInputStream and java.io.DataOutputStream.

This chapter defines its own set of data types representing class file data: The types u1, u2, and u4 represent an unsigned one-, two-, or four-byte quantity, respectively. In the Java platform, these types may be read by methods such as readUnsignedByte, readUnsignedShort, and readInt of the interface java.io.DataInput.

This chapter presents the class file format using pseudostructures written in a C-like structure notation. To avoid confusion with the fields of classes and class instances, etc., the contents of the structures describing the class file format are referred to as items. Successive items are stored in the class file sequentially, without padding or alignment.

Tables, consisting of zero or more variable-sized items, are used in several class file structures. Although we 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 translate a table index directly to a byte offset into the table.
Where we refer to a data structure as an array, it consists of zero or more contiguous fixed-sized items and can be indexed like an array.

**Note:** We use this font for Prolog code and code fragments, and this font for Java virtual machine instructions and class file structures. Commentary, designed to clarify the specification, is given as indented text between horizontal lines:

Commentary provides intuition, motivation, rationale, examples, etc.

### 4.1 The ClassFile Structure

A class file consists of a single ClassFile structure:

```java
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items in the ClassFile structure are as follows:
THE ClassFile STRUCTURE

magic
The magic item supplies the magic number identifying the class file format; it has the value 0xCAFEBAEB.

minor_version, major_version
The values of the minor_version and major_version items are the minor and major version numbers of this class file. Together, a major and a minor version number determine the version of the class file format. If a class file has major version number $M$ and minor version number $m$, we denote the version of its class file format as $M.m$. Thus, class file format versions may be ordered lexicographically, for example, $1.5 < 2.0 < 2.1$.

A Java virtual machine implementation can support a class file format of version $v$ if and only if $v$ lies in some contiguous range $M_i.0 \leq v \leq M_j.m$. The release level of the Java platform to which a Java virtual machine implementation conforms is responsible for determining the range.\(^1\)

constant_pool_count
The value of the constant_pool_count item is equal to the number of entries in the constant_pool table plus one. A constant_pool index is considered valid if it is greater than zero and less than constant_pool_count, with the exception for constants of type long and double noted in §4.4.5.

constant_pool[]
The constant_pool is a table of structures (§4.4) representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant_pool table entry is indicated by its first “tag” byte.

The constant_pool table is indexed from 1 to constant_pool_count-1.

---

\(^1\) Sun's Java virtual machine implementation in JDK release 1.0.2 supports class file format versions 45.0 through 45.3 inclusive. JDK releases 1.1.X support class file format versions in the range 45.0 through 45.65535 inclusive. For $k \geq 2$, implementations of version 1.$k$ of the Java platform support class file format versions in the range 45.0 through 44+$k$.0 inclusive.
access_flags

The value of the access_flags item is a mask of flags used to denote access permissions to and properties of this class or interface. The interpretation of each flag, when set, is as shown in Table 4.1.

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Declared public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Declared final; no subclasses allowed.</td>
</tr>
<tr>
<td>ACC_SUPER</td>
<td>0x0020</td>
<td>Treat superclass methods specially when invoked by the invokespecial instruction.</td>
</tr>
<tr>
<td>ACC_INTERFACE</td>
<td>0x0200</td>
<td>Is an interface, not a class.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Declared abstract; must not be instantiated.</td>
</tr>
<tr>
<td>ACC_SYNTHETIC</td>
<td>0x1000</td>
<td>Declared synthetic; not present in the source code.</td>
</tr>
<tr>
<td>ACC_ANNOTATION</td>
<td>0x2000</td>
<td>Declared as an annotation type.</td>
</tr>
<tr>
<td>ACC_ENUM</td>
<td>0x4000</td>
<td>Declared as an enum type.</td>
</tr>
</tbody>
</table>

A class may be marked with the ACC_SYNTHETIC flag to indicate that it was generated by the compiler and does not appear in the source code.

A class that represents an anonymous class in the Java programming language (JLS3 §15.9.5) must not have its ACC_FINAL flag set.

The ACC_ENUM flag indicates that this class or its superclass is declared as an enumerated type.

An interface is distinguished by its ACC_INTERFACE flag being set. If its ACC_INTERFACE flag is not set, this class file defines a class, not an interface.

If the ACC_INTERFACE flag of this class file is set, its ACC_ABSTRACT flag must also be set (JLS3 §9.1.1.1). Such a
cLass file must not have its ACC_FINAL, ACC_SUPER or ACC_ENUM flags set.

An annotation type must have its ACC_ANNOTATION flag set. If the ACC_ANNOTATION flag is set, the ACC_INTERFACE flag must be set as well. If the ACC_INTERFACE flag of this class file is not set, it may have any of the other flags in Table 4.1 set, except the ACC_ANNOTATION flag. However, such a class file cannot have both its ACC_FINAL and ACC_ABSTRACT flags set (JLS3 §8.1.1.2).

The ACC_SUPER flag indicates which of two alternative semantics is to be expressed by the invokespecial instruction if it appears in this class. Compilers to the instruction set of the Java virtual machine should set the ACC_SUPER flag.²

All bits of the access_flags item not assigned in Table 4.1 are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

this_class

The value of the this_class item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing the class or interface defined by this class file.

super_class

For a class, the value of the super_class item either must be zero or must be a valid index into the constant_pool table. If the value of the super_class item is nonzero, the constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing the direct superclass of the class defined by this class file. Neither the direct superclass nor any of its superclasses may have the ACC_FINAL flag set in the access_flags item of its ClassFile structure.

² The ACC_SUPER flag exists for backward compatibility with code compiled by Sun’s older compilers for the Java programming language. In Sun’s JDK prior to release 1.0.2, the compiler generated ClassFile access_flags in which the flag now representing ACC_SUPER had no assigned meaning, and Sun’s Java virtual machine implementation ignored the flag if it was set.
If the value of the super_class item is zero, then this class file must represent the class Object, the only class or interface without a direct superclass.

For an interface, the value of the super_class item must always be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info structure representing the class Object.

interfaces_count
The value of the interfaces_count item gives the number of direct superinterfaces of this class or interface type.

interfaces[]
Each value in the interfaces array must be a valid index into the constant_pool table. The constant_pool entry at each value of interfaces[i], where 0 ≤ i < interfaces_count, must be a CONSTANT_Class_info (§4.4.1) structure representing an interface that is a direct superinterface of this class or interface type, in the left-to-right order given in the source for the type.

fields_count
The value of the fields_count item gives the number of field_info structures in the fields table. The field_info (§4.5) structures represent all fields, both class variables and instance variables, declared by this class or interface type.

fields[]
Each value in the fields table must be a field_info (§4.5) structure giving a complete description of a field in this class or interface. The fields table includes only those fields that are declared by this class or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

methods_count
The value of the methods_count item gives the number of method_info structures in the methods table.

methods[]
Each value in the methods table must be a method_info (§4.6) structure giving a complete description of a method in this class
or interface. If neither of the ACC_NATIVE and ACC_ABSTRACT flags are set in the access_flags item of a method_info structure, the Java virtual machine instructions implementing the method are also supplied.

The method_info structures represent all methods declared by this class or interface type, including instance methods, class methods, instance initialization methods (§3.9), and any class or interface initialization method (§3.9). The methods table does not include items representing methods that are inherited from superclasses or superinterfaces.

attributes_count

The value of the attributes_count item gives the number of attributes (§4.7) in the attributes table of this class.

attributes[]

Each value of the attributes table must be an attribute_info structure (§4.7).

The only attributes defined by this specification as appearing in the attributes table of a ClassFile structure are the InnerClasses (§4.7.6), EnclosingMethod (§4.7.7), Synthetic (§4.7.8), Signature (§4.7.9), SourceFile (§4.7.10), SourceDebugExtension (§4.7.11), Deprecated (§4.7.15), RuntimeVisibleAnnotations (§4.7.16) and RuntimeInvisibleAnnotations (§4.7.17) attributes.

If a Java virtual machine implementation recognizes class files whose version number is 49.0 or above, it must recognize and correctly read Signature (§4.7.9), RuntimeVisibleAnnotations (§4.7.16), and RuntimeInvisibleAnnotations (§4.7.17) attributes found in the attributes table of a ClassFile structure of a class file whose version number is 49.0 or above.

A Java virtual machine implementation is required to silently ignore any or all attributes in the attributes table of a ClassFile structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).
4.2 The Internal Form of Names

4.2.1 Binary Class and Interface Names

Class and interface names that appear in class file structures are always represented in a fully qualified form as binary names (JLS3 §13.1). Such names are always represented as CONSTANT_Utf8_info (§4.4.7) structures and thus may be drawn, where not further constrained, from the entire Unicode character set. Class and interface names are referenced from those CONSTANT_NameAndType_info (§4.4.6) structures which have such names as part of their descriptor (§4.3), and from all CONSTANT_Class_info (§4.4.1) structures.

For historical reasons, the syntax of binary names that appear in class file structures differs from the syntax of binary names documented in JLS §13.1. In this internal form, the ASCII periods ('.') that normally separate the identifiers that make up the binary name are replaced by ASCII forward slashes ('/'). The identifiers themselves must be unqualified names (§4.2.2).

For example, the normal binary name of class Thread is java.lang.Thread. In the internal form used in descriptors in the class file format, a reference to the name of class Thread is implemented using a CONSTANT_Utf8_info structure representing the string "java/lang/Thread".

4.2.2 Unqualified Names

Names of methods, fields and local variables are stored as unqualified names. Unqualified names must not contain the characters '.', ';', '[' or '/'. Method names are further constrained so that, with the exception of the special method names <init> and <clinit> (§3.9), they must not contain the characters '<' or '>'.

4.3 Descriptors and Signatures

A descriptor is a string representing the type of a field or method. Descriptors are represented in the class file format using modified UTF-8 strings (§4.4.7) and thus may be drawn, where not further constrained, from the entire Unicode character set.

A signature is a string representing the generic type of a field or method, or generic type information for a class declaration.
4.3.1 Grammar Notation

Descriptors and signatures are specified using a grammar. This grammar is a set of productions that describe how sequences of characters can form syntactically correct descriptors of various types. Terminal symbols of the grammar are shown in **bold fixed-width** font. Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined, followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the production:

\[
\text{FieldType:} \\
\text{ BaseType} \\
\text{ ObjectType} \\
\text{ ArrayType}
\]

states that a *FieldType* may represent either a *BaseType*, an *ObjectType* or an *ArrayType*.

A nonterminal symbol on the right-hand side of a production that is followed by an asterisk (*) represents zero or more possibly different values produced from that nonterminal, appended without any intervening space. Similarly, a nonterminal symbol on the right-hand side of a production that is followed by a plus sign (+) represents one or more possibly different values produced from that nonterminal, appended without any intervening space. The production:

\[
\text{MethodDescriptor:} \\
( \text{ ParameterDescriptor*}) \text{ ReturnDescriptor}
\]

states that a *MethodDescriptor* represents a left parenthesis, followed by zero or more *ParameterDescriptor* values, followed by a right parenthesis, followed by a *ReturnDescriptor*.

4.3.2 Field Descriptors

A *field descriptor* represents the type of a class, instance, or local variable. It is a series of characters generated by the grammar:

\[
\text{FieldDescriptor:} \\
\text{ FieldType} \\
\text{ ComponentType:} \\
\text{ FieldType}
\]
**FieldType:**
- **BaseType**
- **ObjectType**
- **ArrayType**

**BaseType:**
- B
- C
- D
- F
- I
- J
- S
- Z

**ObjectType:**
- L Classname;

**ArrayType:**
- [ComponentType

The characters of **BaseType**, the **L** and **;** of **ObjectType**, and the **[** of **ArrayType** are all ASCII characters. The **Classname** represents a binary class or interface name encoded in internal form (§4.2.1). A type descriptor representing an array type is valid only if it represents a type with 255 or fewer dimensions. The interpretation of the field types is as shown in Table 4.2.

<table>
<thead>
<tr>
<th><strong>BaseType</strong> Character</th>
<th><strong>Type</strong></th>
<th><strong>Interpretation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>byte</td>
<td>signed byte</td>
</tr>
<tr>
<td>C</td>
<td>char</td>
<td>Unicode character</td>
</tr>
<tr>
<td>D</td>
<td>double</td>
<td>double-precision floating-point value</td>
</tr>
<tr>
<td>F</td>
<td>float</td>
<td>single-precision floating-point value</td>
</tr>
<tr>
<td>I</td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>J</td>
<td>long</td>
<td>long integer</td>
</tr>
<tr>
<td>L Classname;</td>
<td>reference</td>
<td>an instance of class &lt;classname&gt;</td>
</tr>
<tr>
<td>S</td>
<td>short</td>
<td>signed short</td>
</tr>
<tr>
<td>Z</td>
<td>boolean</td>
<td>true or false</td>
</tr>
<tr>
<td>[</td>
<td>reference</td>
<td>one array dimension</td>
</tr>
</tbody>
</table>
For example, the descriptor of an instance variable of type \texttt{int} is simply \texttt{I}. The descriptor of an instance variable of type \texttt{Object} is \texttt{Ljava/lang/Object;}. Note that the internal form of the binary name for class \texttt{Object} is used. The descriptor of an instance variable that is a multidimensional \texttt{double} array,

\begin{verbatim}
double d[][][];
\end{verbatim}

is

\texttt{[[D}

### 4.3.3 Method Descriptors

A \textit{method descriptor} represents the parameters that the method takes and the value that it returns:

\begin{verbatim}
MethodDescriptor: ( ParameterDescriptor* ) ReturnDescriptor
\end{verbatim}

A \textit{parameter descriptor} represents a parameter passed to a method:

\begin{verbatim}
ParameterDescriptor: FieldType
\end{verbatim}

A \textit{return descriptor} represents the type of the value returned from a method. It is a series of characters generated by the grammar:

\begin{verbatim}
ReturnDescriptor: FieldType VoidDescriptor
\end{verbatim}

\begin{verbatim}
VoidDescriptor: V
\end{verbatim}

The character \texttt{V} indicates that the method returns no value (its return type is \texttt{void}).

A method descriptor is valid only if it represents method parameters with a total length of 255 or less, where that length includes the contribution for \texttt{this} in the case of instance or interface method invocations. The total length is calculated by summing the contributions of the individual parameters, where a parameter of type \texttt{long} or \texttt{double} contributes two units to the length and a parameter of any other type contributes one unit.
For example, the method descriptor for the method

Object mymethod(int i, double d, Thread t)

is

(IDLjava/lang/Thread;)Ljava/lang/Object;

Note that the internal forms of the binary names of Thread and Object are used in the method descriptor.

The method descriptor for mymethod is the same whether mymethod is a class or an instance method. Although an instance method is passed this, a reference to the current class instance, in addition to its intended parameters, that fact is not reflected in the method descriptor. (A reference to this is not passed to a class method.) The reference to this is passed implicitly by the method invocation instructions of the Java virtual machine used to invoke instance methods.

### 4.3.4 Signatures

Signatures are used to encode Java programming language type information that is not part of the Java virtual machine type system, such as generic type and method declarations and parameterized types. See *The Java Language Specification, Third Edition*, for details about such types.

---

This kind of type information is needed to support reflection and debugging, and by the Java compiler.

---

In the following, the terminal symbol **Identifier** is used to denote an identifier for a type, field, local variable, parameter, method name or type variable, as generated by the Java compiler. Such an identifier may contain characters that must not appear in a legal identifier in the Java programming language.

A class signature, defined by the production **ClassSignature**, is used to encode type information about a class declaration. It describes any formal type parameters the class might have, and lists its (possibly parameterized) direct superclass and direct superinterfaces, if any.

**ClassSignature:**

*FormalTypeParametersopt SuperclassSignature SuperinterfaceSignature*
A formal type parameter is described by its name, followed by its class and interface bounds. If the class bound does not specify a type, it is taken to be `Object`.

### FormalTypeParameters:
- `<FormalTypeParameter+>`

### FormalTypeParameter:
- `Identifier ClassBound InterfaceBound*`

### ClassBound:
- `: FieldTypeSignatureopt`

### InterfaceBound:
- `: FieldTypeSignature`

### SuperclassSignature:
- `ClassTypeSignature`

### SuperinterfaceSignature:
- `ClassTypeSignature`

A field type signature, defined by the production `FieldTypeSignature`, encodes the (possibly parameterized) type for a field, parameter or local variable.

### FieldTypeSignature:
- `ClassTypeSignature`
- `ArrayTypeSignature`
- `TypeVariableSignature`

A class type signature gives complete type information for a class or interface type. The class type signature must be formulated such that it can be reliably mapped to the binary name of the class it denotes by erasing any type arguments and converting each ‘.’ character in the signature to a ‘$’ character.

### ClassTypeSignature:
- `L PackageSpecifieropt SimpleClassTypeSignature ClassTypeSignatureSuffix* ;`

### PackageSpecifier:
- `Identifier / PackageSpecifier*`

### SimpleClassTypeSignature:
- `Identifier TypeArgumentsopt`
A method signature, defined by the production `MethodTypeSignature`, encodes the (possibly parameterized) types of the method’s formal arguments and of the exceptions it has declared in its throws clause, its (possibly parameterized) return type, and any formal type parameters in the method declaration.

`MethodTypeSignature:`
`FormalTypeParametersopt (TypeSignature*) ReturnType ThrowsSignature*`

`ReturnType:`
`TypeSignature`
`VoidDescriptor`

`ThrowsSignature:`
`^ClassTypeSignature`
`^TypeVariableSignature`

If the throws clause of a method or constructor does not involve type variables, the `ThrowsSignature` may be elided from the `MethodTypeSignature`. 
A Java compiler must output generic signature information for any class, interface, constructor or member whose generic signature in the Java programming language would include references to type variables or parameterized types.

The signature and descriptor (§4.3.3) of a given method or constructor may not correspond exactly, due to compiler-generated artifacts. In particular, the number of TypeSignatures that encode formal arguments in MethodTypeSignature may be less than the number of ParameterDescriptors in MethodDescriptor.

Sun’s Java virtual machine implementation does not check the well formedness of the signatures described in this subsection during loading or linking. Instead, these checks are deferred until the signatures are used by reflective methods, as specified in the API of Class and members of java.lang.reflect. Future versions of a Java virtual machine implementation may be required to perform some or all of these checks during loading or linking.

4.4 The Constant Pool

Java virtual machine instructions do not rely on the runtime layout of classes, interfaces, class instances, or arrays. Instead, instructions refer to symbolic information in the constant_pool table.

All constant_pool table entries have the following general format:

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

Each item in the constant_pool table must begin with a 1-byte tag indicating the kind of cp_info entry. The contents of the info array vary with the value of tag. The valid tags and their values are listed in Table 4.3. 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.
4.4.1 The CONSTANT_Class_info Structure

The CONSTANT_Class_info structure is used to represent a class or an interface:

```java
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

The items of the CONSTANT_Class_info structure are the following:

- **tag**
  - The `tag` item has the value CONSTANT_Class (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 (§4.4.7) structure representing a valid binary class or interface name encoded in internal form (§4.2.1).
Because arrays are objects, the opcodes `anewarray` and `multianewarray` can reference array “classes” via CONSTANT_Class_info (§4.4.1) structures in the constant_pool table. For such array classes, the name of the class is the descriptor of the array type.

For example, the class name representing a two-dimensional `int` array type `int[][]` is `[[I`.

The class name representing the type array of class `Thread` `Thread[]` is `[Ljava/lang/Thread;`

An array type descriptor is valid only if it represents 255 or fewer dimensions.

**4.4.2 The CONSTANT_Fieldref_info, CONSTANT_Methodref_info, and CONSTANT_InterfaceMethodref_info Structures**

Fields, methods, and interface methods are represented by similar structures:

```java
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_InterfaceMethodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```
The items of these structures are as follows:

**tag**

The tag item of a `CONSTANT_Fieldref_info` structure has the value `CONSTANT_Fieldref (9).`

The tag item of a `CONSTANT_Methodref_info` structure has the value `CONSTANT_Methodref (10).`

The tag item of a `CONSTANT_InterfaceMethodref_info` structure has the value `CONSTANT_InterfaceMethodref (11).`

**class_index**

The value of the class_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a `CONSTANT_Class_info` (§4.4.1) structure representing a class or interface type that has the field or method as a member.

The class_index item of a `CONSTANT_Methodref_info` structure must be a class type, not an interface type. The class_index item of a `CONSTANT_InterfaceMethodref_info` structure must be an interface type. The class_index item of a `CONSTANT_Fieldref_info` structure may be either a class type or an interface type.

**name_and_type_index**

The value of the name_and_type_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a `CONSTANT_NameAndType_info` (§4.4.6) structure. This constant_pool entry indicates the name and descriptor of the field or method. In a `CONSTANT_Fieldref_info` the indicated descriptor must be a field descriptor (§4.3.2). Otherwise, the indicated descriptor must be a method descriptor (§4.3.3).

If the name of the method of a `CONSTANT_Methodref_info` structure begins with a ' <' ('<'), then the name must be the special name `<init>`, representing an instance initialization method (§3.9). The return type of such a method must be `void`. 
4.4.3 The CONSTANT_String_info Structure

The CONSTANT_String_info structure is used to represent constant objects of the type String:

```
CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

The items of the CONSTANT_String_info structure are as follows:

- **tag**
  
  The tag item of the CONSTANT_String_info structure has the value CONSTANT_String (8).

- **string_index**
  
  The value of the string_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 (§4.4.7) structure representing the sequence of characters to which the String object is to be initialized.

4.4.4 The CONSTANT_Integer_info and CONSTANT_Float_info Structures

The CONSTANT_Integer_info and CONSTANT_Float_info structures represent 4-byte numeric (int and float) constants:

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

CONSTANT_Float_info {
    u1 tag;
    u4 bytes;
}
```

The items of these structures are as follows:
tag

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

The tag item of the CONSTANT_Float_info structure has the value CONSTANT_Float (4).

bytes

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

The bytes item of the CONSTANT_Float_info structure represents the value of the float constant in IEEE 754 floating-point single format (§3.3.2). The bytes of the single format representation are stored in big-endian (high byte first) order.

The value represented by the CONSTANT_Float_info structure is determined as follows. The bytes of the value are first converted into an int constant bits. Then:

- If bits is 0x7f800000, the float value will be positive infinity.
- If bits is 0xff800000, the float value will be negative infinity.
- If bits is in the range 0x7f800001 through 0x7fffffff or in the range 0xff800001 through 0xffffffff, the float value will be NaN.
- In all other cases, let s, e, and m be three values that might be computed from bits:

\[
\begin{align*}
\text{int } s &= ((\text{bits} >> 31) == 0) ? 1 : -1; \\
\text{int } e &= ((\text{bits} >> 23) & 0xff); \\
\text{int } m &= (e == 0) ? \\
&\quad (\text{bits} & 0x7fffffff) << 1 : \\
&\quad (\text{bits} & 0x7fffffff) \mid 0x800000;
\end{align*}
\]

Then the float value equals the result of the mathematical expression \( s \cdot m \cdot 2^{e-150} \).
4.4.5 The CONSTANT_Long_info and CONSTANT_Double_info Structures

The CONSTANT_Long_info and CONSTANT_Double_info represent 8-byte numeric (long and double) constants:

```
CONSTANT_Long_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

```
CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}
```

All 8-byte constants take up two entries in the constant_pool table of the class file. If a CONSTANT_Long_info or CONSTANT_Double_info structure is the item in the constant_pool table at index n, then the next usable item in the pool is located at index n + 2. The constant_pool index n + 1 must be valid but is considered unusable.3

The items of these structures are as follows:

tag
The tag item of the CONSTANT_Long_info structure has the value CONSTANT_Long (5).

The tag item of the CONSTANT_Double_info structure has the value CONSTANT_Double (6).

high_bytes, low_bytes
The unsigned high_bytes and low_bytes items of the CONSTANT_Long_info structure together represent the value of the long constant ((long) high_bytes << 32) + low_bytes, where the bytes of each of high_bytes and low_bytes are stored in big-endian (high byte first) order.

---

3 In retrospect, making 8-byte constants take two constant pool entries was a poor choice.
The high_bytes and low_bytes items of the CONSTANT_Double_info structure together represent the double value in IEEE 754 floating-point double format (§3.3.2). The bytes of each item are stored in big-endian (high byte first) order.

The value represented by the CONSTANT_Double_info structure is determined as follows. The high_bytes and low_bytes items are converted into the long constant bits, which is equal to ((long) high_bytes << 32) + low_bytes. Then:

- If bits is 0x7ff0000000000000L, the double value will be positive infinity.
- If bits is 0xffff000000000000L, the double value will be negative infinity.
- If bits is in the range 0x7ff0000000000001L through 0x7fffffffffffffffL or in the range 0xfff0000000000001L through 0xffffffffffffffffL, the double value will be NaN.
- In all other cases, let s, e, and m be three values that might be computed from bits:

  int s = ((bits >> 63) == 0) ? 1 : -1;
  int e = (int)(((bits >> 52) & 0x7ffL);
  long m = (e == 0) ?
    (bits & 0xfffffffffffffffL) << 1 :
    (bits & 0xfffffffffffffffL) | 0x100000000000000L;

  Then the floating-point value equals the double value of the mathematical expression $s \cdot m \cdot 2^e = 1075$.

4.4.6 The CONSTANT_NameAndType_info Structure

The CONSTANT_NameAndType_info structure is used to represent a field or method, without indicating which class or interface type it belongs to:

CONSTANT_NameAndType_info {
  u1 tag;
  u2 name_index;
  u2 descriptor_index;
}
The items of the CONSTANT_NameAndType_info structure are as follows:

**tag**

The tag item of the CONSTANT_NameAndType_info structure has the value CONSTANT_NameAndType (12).

**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 (§4.4.7) structure representing either the special method name <init> (§3.9) or a valid unqualified name (§4.2.2) denoting a field or method.

**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 (§4.4.7) structure representing a valid field descriptor (§4.3.2) or method descriptor (§4.3.3).

### 4.4.7 The CONSTANT_Utf8_info Structure

The CONSTANT_Utf8_info structure is used to represent constant string values:

```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 of the CONSTANT_Utf8_info structure has the value 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 lie in the range (byte)0xf0-(byte)0xff.

String content is encoded in modified UTF-8. Modified UTF-8 strings are encoded so that character sequences that contain only non-null ASCII characters can be represented using only 1 byte per character, but all Unicode characters can be represented.

Characters in the range '\u0001' to '\u007F' are represented by a single byte:

```
0  bits 6-0
```

The 7 bits of data in the byte give the value of the character represented.

The null character ('\u0000') and characters in the range '\u0080' to '\u07FF' are represented by a pair of bytes x and y:

```
x: 1 1 0  bits 10-6  y: 1 0  bits 5-0
```

The bytes represent the character with the value:

\[(x \& 0x1f) \ll 6 + (y \& 0x3f)\]

Characters in the range '\u0800' to '\uFFFF' are represented by 3 bytes x, y, and z:

```
x: 1 1 1 0  bits 15-12  y: 1 0  bits 11-6  z: 1 0  bits 5-0
```

The three bytes represent the character with the value:

\[(x \& 0xf) \ll 12 + (y \& 0x3f) \ll 6 + (z \& 0x3f)\]

Characters with code points above U+FFFF (so-called supplementary characters) are represented by separately encoding the two surrogate code units of their UTF-16 representation. Each of the surrogate code units is represented by three bytes. This means supplementary characters are represented by six bytes, u, v, w, x, y, and z:

```
u: 1 1 1 0 1 1 0 1  v: 1 0 1 0 (bits 20-16)-1  w: 1 0  bits 15-10
x: 1 1 1 0 1 1 0 1  y: 1 0 1 1  bits 9-6  z: 1 0  bits 5-0
```
The six bytes represent the character with the value:

\[0x10000 + ((v & 0x0f) << 16) + ((w & 0x3f) << 10) + (y & 0x0f) << 6) + (z & 0x3f)\]

The bytes of multibyte characters are stored in the class file in big-endian (high byte first) order.

There are two differences between this format and the “standard” UTF-8 format. First, the null character (char) 0 is encoded using the 2-byte format rather than the 1-byte format, so that modified UTF-8 strings never have embedded nulls. Second, only the 1-byte, 2-byte, and 3-byte formats of standard UTF-8 are used. The Java virtual machine does not recognize the four-byte format of standard UTF-8; it uses its own two-times-three-byte format instead.

For more information regarding the standard UTF-8 format, see Section 3.9 Unicode Encoding Forms of The Unicode Standard, Version 4.0.

### 4.5 Fields

Each field is described by a field_info structure. No two fields in one class file may have the same name and descriptor (§4.3.2).

The structure has the following format:

```c
field_info {
    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:

access_flags

The value of the access_flags item is a mask of flags used to denote access permission to and properties of this field. The interpretation of each flag, when set, is as shown in Table 4.4.
A field may be marked with the ACC_SYNTHETIC flag to indicate that it was generated by the compiler and does not appear in the source code.

The ACC_ENUM flag indicates that this field is used to hold an element of an enumerated type.

Fields of classes may set any of the flags in Table 4.4. However, a specific field of a class may have at most one of its ACC_PRIVATE, ACC_PROTECTED, and ACC_PUBLIC flags set (JLS3 §8.3.1) and must not have both its ACC_FINAL and ACC_VOLATILE flags set (JLS3 §8.3.1.4).

All fields of interfaces must have their ACC_PUBLIC, ACC_STATIC, and ACC_FINAL flags set; they may have their ACC_SYNTHETIC flag set and must not have any of the other flags in Table 4.4 set (JLS3 §9.3).

All bits of the access_flags item not assigned in Table 4.4 are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

### Table 4.4 Field access and property flags

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Declared public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
<td>Declared private; usable only within the defining class.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Declared protected; may be accessed within subclasses.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Declared static.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Declared final; never directly assigned to after object construction (JLS3 §17.5).</td>
</tr>
<tr>
<td>ACC_VOLATILE</td>
<td>0x0040</td>
<td>Declared volatile; cannot be cached.</td>
</tr>
<tr>
<td>ACC_TRANSIENT</td>
<td>0x0080</td>
<td>Declared transient; not written or read by a persistent object manager.</td>
</tr>
<tr>
<td>ACC_SYNTHETIC</td>
<td>0x1000</td>
<td>Declared synthetic; not present in the source code.</td>
</tr>
<tr>
<td>ACC_ENUM</td>
<td>0x4000</td>
<td>Declared as an element of an enum.</td>
</tr>
</tbody>
</table>
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 (§4.4.7) structure which must represent a valid unqualified name (§4.2.2) denoting a field.

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 (§4.4.7) structure that must represent a valid field descriptor (§4.3.2).

attributes_count
The value of the attributes_count item indicates the number of additional attributes (§4.7) of this field.

attributes[]
Each value of the attributes table must be an attribute structure (§4.7). A field can have any number of attributes associated with it.

The attributes defined by this specification as appearing in the attributes table of a field_info structure are ConstantValue (§4.7.2), Synthetic (§4.7.8), Signature (§4.7.9), Deprecated (§4.7.15), RuntimeVisibleAnnotations (§4.7.16) and RuntimeInvisibleAnnotations (§4.7.17).

A Java virtual machine implementation must recognize and correctly read ConstantValue (§4.7.2) attributes found in the attributes table of a field_info structure. If a Java virtual machine implementation recognizes class files whose version number is 49.0 or above, it must recognize and correctly read Signature (§4.7.9), RuntimeVisibleAnnotations (§4.7.16) and RuntimeInvisibleAnnotations (§4.7.17) attributes found in the attributes table of a field_info structure of a class file whose version number is 49.0 or above.

A Java virtual machine implementation is required to silently ignore any or all attributes that it does not recognize in the attributes table of a field_info structure. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).
4.6 Methods

Each method, including each instance initialization method (§3.9) and the class or interface initialization method (§3.9), is described by a `method_info` structure. No two methods in one class file may have the same name and descriptor (§4.3.3).

The structure has the following format:

```java
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

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

`access_flags`

The value of the `access_flags` item is a mask of flags used to denote access permission to and properties of this method. The interpretation of each flag, when set, is as shown in Table 4.5.

The ACC_VARARGS flag indicates that this method takes a variable number of arguments at the source code level. A method declared to take a variable number of arguments must be compiled with the ACC_VARARGS flag set to 1. All other methods must be compiled with the ACC_VARARGS flag set to 0.

The ACC_BRIDGE flag is used to indicate a bridge method generated by the compiler.

A method may be marked with the ACC_SYNTHETIC flag to indicate that it was generated by the compiler and does not appear in the source code, unless it is one of the methods named in Section 4.7.8, “The Synthetic Attribute”.

Methods of classes may set any of the flags in Table 4.5. However, a specific method of a class may have at most one of its ACC_PRIVATE, ACC_PROTECTED and ACC_PUBLIC flags set (JLS3 §8.4.3). If a specific method has its ACC_ABSTRACT flag set, it must not have any of its ACC_FINAL, ACC_NATIVE, ACC_PRIVATE, ACC_STATIC, ACC STRICT or ACC_SYNCHRONIZED flags set (JLS3 §8.4.3.1, JLS3 §8.4.3.3, JLS3 §8.4.3.4).
### Table 4.5  Method access and property flags

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Declared public; may be accessed from outside its package.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
<td>Declared private; accessible only within the defining class.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Declared protected; may be accessed within subclasses.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Declared static.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Declared final; must not be overridden.</td>
</tr>
<tr>
<td>ACC_SYNCHRONIZED</td>
<td>0x0020</td>
<td>Declared synchronized; invocation is wrapped by a monitor use.</td>
</tr>
<tr>
<td>ACC_BRIDGE</td>
<td>0x0040</td>
<td>A bridge method, generated by the compiler.</td>
</tr>
<tr>
<td>ACC_VARARGS</td>
<td>0x0080</td>
<td>Declared with variable number of arguments.</td>
</tr>
<tr>
<td>ACC_NATIVE</td>
<td>0x0100</td>
<td>Declared native; implemented in a language other than Java.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Declared abstract; no implementation is provided.</td>
</tr>
<tr>
<td>ACC_STRICT</td>
<td>0x0800</td>
<td>Declared strictfp; floating-point mode is FP-strict</td>
</tr>
<tr>
<td>ACC_SYNTHETIC</td>
<td>0x1000</td>
<td>Declared synthetic; not present in the source code.</td>
</tr>
</tbody>
</table>

All interface methods must have their ACC_ABSTRACT and ACC_PUBLIC flags set; they may have their ACC_VARARGS, ACC_BRIDGE and ACC_SYNTHETIC flags set and must not have any of the other flags in Table 4.5 set (JLS3 §9.4).

A specific instance initialization method (§3.9) may have at most one of its ACC_PRIVATE, ACC_PROTECTED, and ACC_PUBLIC flags set, and may also have its ACC_STRICT, ACC_VARARGS and ACC_SYNTHETIC flags set, but must not have any of the other flags in Table 4.5 set.
Class and interface initialization methods (§3.9) are called implicitly by the Java virtual machine. The value of their access_flags item is ignored except for the setting of the ACC_STRICT flag.

All bits of the access_flags item not assigned in Table 4.5 are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

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 (§4.4.7) structure representing either one of the special method names (§3.9) <init> or <clinit>, or a valid unqualified name (§4.2.2) denoting a method.

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 (§4.4.7) structure representing a valid method descriptor (§4.3.3).

attributes_count

The value of the attributes_count item indicates the number of additional attributes (§4.7) of this method.

attributes[]

Each value of the attributes table must be an attribute structure (§4.7). A method can have any number of optional attributes associated with it.

The only attributes defined by this specification as appearing in the attributes table of a method_info structure are the Code (§4.7.3), Exceptions (§4.7.5), Synthetic (§4.7.8), Signature (§4.7.9), Deprecated (§4.7.15), RuntimeVisibleAnnotations (§4.7.16), RuntimeInvisibleAnnotations (§4.7.17), RuntimeVisibleParameterAnnotations (§4.7.18),
RuntimeInvisibleParameterAnnotations (§4.7.19) and AnnotationDefault (§4.7.20) attributes.

A Java virtual machine implementation must recognize and correctly read Code (§4.7.3) and Exceptions (§4.7.5) attributes found in the attributes table of a method_info structure. If a Java virtual machine implementation recognizes class files whose version number is 49.0 or above, it must recognize and correctly read Signature (§4.7.9), RuntimeVisibleAnnotations (§4.7.16), RuntimeInvisibleAnnotations (§4.7.17), RuntimeVisibleParameterAnnotations (§4.7.18), RuntimeInvisibleParameterAnnotations (§4.7.19) and AnnotationDefault (§4.7.20) attributes found in the attributes table of a method_info structure of a class file whose version number is 49.0 or above.

A Java virtual machine implementation is required to silently ignore any or all attributes in the attributes table of a method_info structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).
4.7 Attributes

Attributes are used in the ClassFile (§4.1), field_info (§4.5), method_info (§4.6) and Code_attribute (§4.7.3) structures of the class file format. All attributes have the following general format:

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

For all attributes, the `attribute_name_index` must be a valid unsigned 16-bit index into the constant pool of the class. The `constant_pool` entry at `attribute_name_index` must be a `CONSTANT_Utf8_info` (§4.4.7) structure representing the name of the attribute. The value of the `attribute_length` item indicates the length of the subsequent information in bytes. The length does not include the initial six bytes that contain the `attribute_name_index` and `attribute_length` items.

Certain attributes are predefined as part of the class file specification. They are listed in Table 4.6, accompanied by the version of the Java Platform, Standard Edition (“Java SE”) and the version of the class file format in which each first appeared. Within the context of their use in this specification, that is, in the attributes tables of the class file structures in which they appear, the names of these predefined attributes are reserved. Of the predefined attributes:

- The `ConstantValue`, `Code` and `Exceptions` attributes must be recognized and correctly read by a class file reader for correct interpretation of the class file by a Java virtual machine implementation.

- The `InnerClasses`, `EnclosingMethod` and `Synthetic` attributes must be recognized and correctly read by a class file reader in order to properly implement the Java platform class libraries (§3.12).

- The `Signature`, `RuntimeVisibleAnnotations`, `RuntimeInvisibleAnnotations`, `RuntimeVisibleParameterAnnotations`, `RuntimeInvisibleParameterAnnotations` and `AnnotationDefault` attributes must be recognized and correctly read by a class file reader if the class file’s version number is 49.0 or above and the Java virtual machine implementation recognizes class files whose version number is 49.0 or above.
The StackMapTable attribute must be recognized and correctly read by a class file reader if the class file’s version number is 50.0 or above and the Java virtual machine implementation recognizes class files whose version number is 50.0 or above.

Use of the remaining predefined attributes is optional; a class file reader may use the information they contain, or otherwise must silently ignore those attributes.
4.7.1 Defining and Naming New Attributes

Compilers are permitted to define and emit class files containing new attributes in the attributes tables of class file structures. Java virtual machine implementations are permitted to recognize and use new attributes found in the attributes tables of class file structures. However, any attribute not defined as part of this Java virtual machine specification must not affect the semantics of class or interface types. Java virtual machine implementations are required to silently ignore attributes they do not recognize.

For instance, defining a new attribute to support vendor-specific debugging is permitted. Because Java virtual machine implementations are required to ignore attributes they do not recognize, class files intended for that particular Java virtual machine implementation will be usable by other implementations even if those implementations cannot make use of the additional debugging information that the class files contain.

Java virtual machine implementations are specifically prohibited from throwing an exception or otherwise refusing to use class files simply because of the presence of some new attribute. Of course, tools operating on class files may not run correctly if given class files that do not contain all the attributes they require.

Two attributes that are intended to be distinct, but that happen to use the same attribute name and are of the same length, will conflict on implementations that recognize either attribute. Attributes defined other than by Sun must have names chosen according to the package naming convention defined by The Java Language Specification (JLS3 §7.7). For instance, a new attribute defined by Netscape might have the name "com.netscape.new-attribute".4

Sun may define additional attributes in future versions of this specification.

4.7.2 The ConstantValue Attribute

The ConstantValue attribute is a fixed-length attribute in the attributes table of a field_info (§4.5) structure. A ConstantValue attribute represents the value of a constant field. There can be no more than one ConstantValue attribute in the

---

4 The first edition of The Java Language Specification required that "com" be in uppercase in this example. The second edition reversed that convention and used lowercase.
attributes table of a given field_info structure. If the field is static (that is, the ACC_STATIC flag (Table 4.4) in the access_flags item of the field_info structure is set) then the constant field represented by the field_info structure is assigned the value referenced by its ConstantValue attribute as part of the initialization of the class or interface declaring the constant field (§5.5). This occurs immediately prior to the invocation of the class or interface initialization method (§3.9) of that class or interface.

If a field_info structure representing a non-static field has a ConstantValue attribute, then that attribute must silently be ignored. Every Java virtual machine implementation must recognize ConstantValue attributes.

The ConstantValue attribute has the following format:

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

The items of the ConstantValue_attribute structure are as follows:

attribute_name_index

The value of the attribute_name_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Utf8_info (§4.4.7) 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 gives the constant value represented by this attribute. The constant_pool entry must be of a type appropriate to the field, as shown by Table 4.7.
Table 4.7 CONSTANT value attribute types

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Entry Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>CONSTANT_Long</td>
</tr>
<tr>
<td>float</td>
<td>CONSTANT_Float</td>
</tr>
<tr>
<td>double</td>
<td>CONSTANT_Double</td>
</tr>
<tr>
<td>int, short, char, byte, boolean</td>
<td>CONSTANT_Integer</td>
</tr>
<tr>
<td>String</td>
<td>CONSTANT_String</td>
</tr>
</tbody>
</table>

4.7.3 The Code Attribute

The Code attribute is a variable-length attribute in the attributes table of a method_info (§4.6) structure. A Code attribute contains the Java virtual machine instructions and auxiliary information for a single method, instance initialization method (§3.9), or class or interface initialization method (§3.9). Every Java virtual machine implementation must recognize Code attributes. If the method is either native or abstract, its method_info structure must not have a Code attribute. Otherwise, its method_info structure must have exactly one Code attribute.

The Code attribute has the following format:

```
Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    { u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

The items of the Code_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 (§4.4.7) structure representing the string "Code".

attribute_length
The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

max_stack
The value of the max_stack item gives the maximum depth (§3.6.2) of the operand stack of this method at any point during execution of the method.

max_locals
The value of the max_locals item gives the number of local variables in the local variable array allocated upon invocation of this method, including the local variables used to pass parameters to the method on its invocation.

The greatest local variable index for a value of type long or double is max_locals−2. The greatest local variable index for a value of any other type is max_locals−1.

code_length
The value of the code_length item gives the number of bytes in the code array for this method. The value of code_length must be greater than zero; the code array must not be empty.

code[]
The code array gives the actual bytes of Java virtual machine code that implement the method.

When the code array is read into memory on a byte-addressable machine, if the first byte of the array is aligned on a 4-byte boundary, the tableswitch and lookupswitch 32-bit offsets will be 4-byte aligned. (Refer to the descriptions of those instructions for more information on the consequences of code array alignment.)

The detailed constraints on the contents of the code array are extensive and are given in a separate section (§4.9).
exception_table_length

The value of the exception_table_length item gives the number of entries in the exception_table table.

exception_table[]

Each entry in the exception_table array describes one exception handler in the code array. The order of the handlers in the exception_table array is significant. See Section 3.10 for more details.

Each exception_table entry contains the following four items:

`start_pc, end_pc`

The values of the two items `start_pc` and `end_pc` indicate the ranges in the code array at which the exception handler is active. The value of `start_pc` must be a valid index into the code array of the opcode of an instruction. The value of `end_pc` either must be a valid index into the code array of the opcode of an instruction or must be equal to `code_length`, the length of the code array. The value of `start_pc` must be less than the value of `end_pc`.

The `start_pc` is inclusive and `end_pc` is exclusive; that is, the exception handler must be active while the program counter is within the interval `[start_pc, end_pc]`.\(^5\)

`handler_pc`

The value of the `handler_pc` item indicates the start of the exception handler. The value of the item must be a valid index into the code array and must be the index of the opcode of an instruction.

---

\(^5\) The fact that `end_pc` is exclusive is a historical mistake in the design of the Java virtual machine: if the Java virtual machine code for a method is exactly 65535 bytes long and ends with an instruction that is 1 byte long, then that instruction cannot be protected by an exception handler. A compiler writer can work around this bug by limiting the maximum size of the generated Java virtual machine code for any method, instance initialization method, or static initializer (the size of any code array) to 65534 bytes.
catch_type

If the value of the catch_type item is nonzero, it must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing a class of exceptions that this exception handler is designated to catch. The exception handler will be called only if the thrown exception is an instance of the given class or one of its subclasses.

If the value of the catch_type item is zero, this exception handler is called for all exceptions. This is used to implement finally (see Section 7.13, “Compiling finally”).

attributes_count

The value of the attributes_count item indicates the number of attributes of the Code attribute.

attributes[]

Each value of the attributes table must be an attribute structure (§4.7). A Code attribute can have any number of optional attributes associated with it.

The only attributes defined by this specification as appearing in the attributes table of a Code attribute are the LineNumberTable (§4.7.12), LocalVariableTable (§4.7.13), LocalVariableTypeTable (§4.7.14), and StackMapTable (§4.7.4) attributes.

If a Java virtual machine implementation recognizes class files whose version number is 50.0 or above, it must recognize and correctly read StackMapTable (§4.7.4) attributes found in the attributes table of a Code attribute of a class file whose version number is 50.0 or above.

A Java virtual machine implementation is required to silently ignore any or all attributes in the attributes table of a Code attribute that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information (§4.7.1).
4.7.4 The StackMapTable Attribute

The StackMapTable attribute is a variable-length attribute in the attributes table of a Code (§4.7.3) attribute. This attribute is used during the process of verification by typechecking (§4.10.1).

A StackMapTable attribute consists of zero or more stack map frames. Each stack map frame specifies (either explicitly or implicitly) a bytecode offset, the verification types (§4.10.1) for the local variables, and the verification types for the operand stack.

The type checker deals with and manipulates the expected types of a method’s local variables and operand stack. Throughout this section, a location refers to either a single local variable or to a single operand stack entry.

We will use the terms stack map frame and type state interchangeably to describe a mapping from locations in the operand stack and local variables of a method to verification types. We will usually use the term stack map frame when such a mapping is provided in the class file, and the term type state when the mapping is used by the type checker.

If a method’s Code attribute does not have a StackMapTable attribute, it has an implicit stack map attribute. This implicit stack map attribute is equivalent to a StackMapTable attribute with number_of_entries equal to zero. A method’s Code attribute may have at most one StackMapTable attribute, otherwise a ClassFormatError is thrown.

The StackMapTable attribute has the following format:

```
StackMapTable_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 number_of_entries;
  stack_map_frame entries[number_of_entries];
}
```

The items of the StackMapTable_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 (§4.4.7) structure representing the string "StackMapTable".
attribute_length
The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

number_of_entries
The value of the number_of_entries item gives the number of stack_map_frame entries in the entries table.

entries
The entries array gives the method's stack_map_frame structures.

Each stack_map_frame structure specifies the type state at a particular bytecode offset. Each frame type specifies (explicitly or implicitly) a value, offset_delta, that is used to calculate the actual bytecode offset at which a frame applies. The bytecode offset at which a frame applies is calculated by adding offset_delta + 1 to the bytecode offset of the previous frame, unless the previous frame is the initial frame of the method, in which case the bytecode offset is offset_delta.

By using an offset delta rather than the actual bytecode offset we ensure, by definition, that stack map frames are in the correctly sorted order. Furthermore, by consistently using the formula offset_delta + 1 for all explicit frames, we guarantee the absence of duplicates.

We say that an instruction in the bytecode has a corresponding stack map frame if the instruction starts at offset \( i \) in the code array of a Code attribute, and the Code attribute has a StackMapTable attribute whose entries array has a stack_map_frame structure that applies at bytecode offset \( i \).

The stack_map_frame structure consists of a one-byte tag followed by zero or more bytes, giving more information, depending upon the tag.

A stack map frame may belong to one of several frame types:
union stack_map_frame {
    same_frame;
    same_locals_1_stack_item_frame;
    same_locals_1_stack_item_frame_extended;
    chop_frame;
    same_frame_extended;
    append_frame;
    full_frame;
}

All frame types, even full_frame, rely on the previous frame for some of their semantics. This raises the question of what is the very first frame? The initial frame is implicit, and computed from the method descriptor. See the Prolog code for methodInitialStackFrame (§4.10.1.3.3).

The frame type same_frame is represented by tags in the range [0-63]. If the frame type is same_frame, it means the frame has exactly the same locals as the previous stack map frame and that the number of stack items is zero. The offset_delta value for the frame is the value of the tag item, frame_type.

    same_frame {
        u1 frame_type = SAME; /* 0-63 */
    }

The frame type same_locals_1_stack_item_frame is represented by tags in the range [64, 127]. If the frame_type is same_locals_1_stack_item_frame, it means the frame has exactly the same locals as the previous stack map frame and that the number of stack items is 1. The offset_delta value for the frame is the value (frame_type - 64). There is a verification_type_info following the frame_type for the one stack item.

    same_locals_1_stack_item_frame {
        u1 frame_type = SAME_LOCALS_1_STACK_ITEM; /* 64-127 */
        verification_type_info stack[1];
    }

Tags in the range [128-246] are reserved for future use.
The frame type `same_locals_1_stack_item_frame_extended` is represented by the tag 247. The frame type `same_locals_1_stack_item_frame_extended` indicates that the frame has exactly the same locals as the previous stack map frame and that the number of stack items is 1. The `offset_delta` value for the frame is given explicitly. There is a `verification_type_info` following the `frame_type` for the one stack item.

```java
same_locals_1_stack_item_frame_extended {
    u1 frame_type = SAME_LOCALS_1_STACK_ITEM_EXTENDED; /* 247 */
    u2 offset_delta;
    verification_type_info stack[1];
}
```

The frame type `chop_frame` is represented by tags in the range [248-250]. If the `frame_type` is `chop_frame`, it means that the operand stack is empty and the current locals are the same as the locals in the previous frame, except that the last `k` locals are absent. The value of `k` is given by the formula `251 - frame_type`.

```java
chop_frame {
    u1 frame_type = CHOP; /* 248-250 */
    u2 offset_delta;
}
```

The frame type `same_frame_extended` is represented by the tag value 251. If the `frame_type` is `same_frame_extended`, it means the frame has exactly the same locals as the previous stack map frame and that the number of stack items is zero.

```java
same_frame_extended {
    u1 frame_type = SAME_FRAME_EXTENDED; /* 251 */
    u2 offset_delta;
}
```

The frame type `append_frame` is represented by tags in the range [252-254]. If the `frame_type` is `append_frame`, it means that the operand stack is empty and the current locals are the same as the locals in the previous frame, except that `k` additional locals are defined. The value of `k` is given by the formula `frame_type - 251`.
append_frame {
    u1 frame_type = APPEND; /* 252-254 */
    u2 offset_delta;
    verification_type_info locals[frame_type - 251];
}

The 0th entry in locals represents the type of the first additional local variable. If locals[M] represents local variable N, then locals[M+1] represents local variable N+1 if locals[M] is one of:

- Top_variable_info
- Integer_variable_info
- Float_variable_info
- Null_variable_info
- UninitializedThis_variable_info
- Object_variable_info
- Uninitialized_variable_info

Otherwise locals[M+1] represents local variable N+2. It is an error if, for any index i, locals[i] represents a local variable whose index is greater than the maximum number of local variables for the method.

The frame type full_frame is represented by the tag value 255.

full_frame {
    u1 frame_type = FULL_FRAME; /* 255 */
    u2 offset_delta;
    u2 number_of_locals;
    verification_type_info locals[number_of_locals];
    u2 number_of_stack_items;
    verification_type_info stack[number_of_stack_items];
}

The 0th entry in locals represents the type of local variable 0. If locals[M] represents local variable N, then locals[M+1] represents local variable N+1 if locals[M] is one of:

- Top_variable_info
- Integer_variable_info
- Float_variable_info
- Null_variable_info
- UninitializedThis_variable_info
- Object_variable_info
- Uninitialized_variable_info

Otherwise, locals[M+1] represents local variable N+2. It is an error if, for any index i, locals[i] represents a local variable whose index is greater than the maximum number of local variables for the method.

The 0th entry in stack represents the type of the bottom of the stack, and subsequent entries represent types of stack elements closer to the top of the operand stack. We shall refer to the bottom element of the stack as stack element 0, and to subsequent elements as stack element 1, 2 etc. If stack[M] represents stack element N, then stack[M+1] represents stack element N+1 if stack[M] is one of:

- Top_variable_info
- Integer_variable_info
- Float_variable_info
- Null_variable_info
- UninitializedThis_variable_info
- Object_variable_info
- Uninitialized_variable_info

Otherwise, stack[M+1] represents stack element N+2. It is an error if, for any index i, stack[i] represents a stack entry whose index is greater than the maximum operand stack size for the method.
The verification_type_info structure consists of a one-byte tag followed by zero or more bytes, giving more information about the tag. Each verification_type_info structure specifies the verification type of one or two locations.

```c
union verification_type_info {
   Top_variable_info;
   Integer_variable_info;
   Float_variable_info;
   Long_variable_info;
   Double_variable_info;
   Null_variable_info;
   UninitializedThis_variable_info;
   Object_variable_info;
   Uninitialized_variable_info;
}
```

The Top_variable_info type indicates that the local variable has the verification type top (∩).

```c
Top_variable_info {
   u1 tag = ITEM_Top; /* 0 */
}
```

The Integer_variable_info type indicates that the location contains the verification type int.

```c
Integer_variable_info {
   u1 tag = ITEM_Integer; /* 1 */
}
```

The Float_variable_info type indicates that the location contains the verification type float.

```c
Float_variable_info {
   u1 tag = ITEM_Float; /* 2 */
}
```

The Long_variable_info type indicates that the location contains the verification type long. If the location is a local variable, then:
• It must not be the local variable with the highest index.

• The next higher numbered local variable contains the verification type $\top$.

  If the location is an operand stack entry, then:

• The current location must not be the topmost location of the operand stack.

• The next location closer to the top of the operand stack contains the verification type $\top$.

This structure gives the contents of two locations in the operand stack or in the local variables.

```
Long_variable_info {
    u1 tag = ITEM_Long; /* 4 */
}
```

The `Double_variable_info` type indicates that the location contains the verification type `double`. If the location is a local variable, then:

• It must not be the local variable with the highest index.

• The next higher numbered local variable contains the verification type $\top$.

  If the location is an operand stack entry, then:

• The current location must not be the topmost location of the operand stack.

• The next location closer to the top of the operand stack contains the verification type $\top$.

This structure gives the contents of two locations in the operand stack or in the local variables.

```
Double_variable_info {
    u1 tag = ITEM_Double; /* 3 */
}
```

The `Null_variable_info` type indicates that location contains the verification type `null`.

```
Null_variable_info {
    u1 tag = ITEM_Null; /* 5 */
}
```
The UninitializedThis_variable_info type indicates that the location contains the verification type uninitializedThis.

```java
UninitializedThis_variable_info {
    u1 tag = ITEM_UninitializedThis; /* 6 */
}
```

The Object_variable_info type indicates that the location contains an instance of the class referenced by the constant pool entry.

```java
Object_variable_info {
    u1 tag = ITEM_Object; /* 7 */
    u2 cpool_index;
}
```

The Uninitialized_variable_info indicates that the location contains the verification type uninitialized(offset). The offset item indicates the offset of the new instruction that created the object being stored in the location.

```java
Uninitialized_variable_info {
    u1 tag = ITEM_Uninitialized /* 8 */
    u2 offset;
}
```

### 4.7.5 The Exceptions Attribute

The Exceptions attribute is a variable-length attribute in the attributes table of a method_info (§4.6) structure. The Exceptions attribute indicates which checked exceptions a method may throw. There may be at most one Exceptions attribute in each method_info structure.

The Exceptions attribute has the following format:

```java
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

The items of the Exceptions_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 the CONSTANT_Utf8_info (§4.4.7) structure representing the string "Exceptions".

attribute_length
The value of the attribute_length item indicates the attribute length, excluding the initial six bytes.

number_of_exceptions
The value of the number_of_exceptions item indicates the number of entries in the exception_index_table.

exception_index_table[]
Each value in the exception_index_table array must be a valid index into the constant_pool table. The constant_pool entry referenced by each table item must be a CONSTANT_Class_info (§4.4.1) structure representing a class type that this method is declared to throw.

A method should throw an exception only if at least one of the following three criteria is met:

- The exception is an instance of RuntimeException or one of its subclasses.
- The exception is an instance of Error or one of its subclasses.
- The exception is an instance of one of the exception classes specified in the exception_index_table just described, or one of their subclasses.

These requirements are not enforced in the Java virtual machine; they are enforced only at compile time.

4.7.6 The InnerClasses Attribute
The InnerClasses attribute\(^6\) is a variable-length attribute in the attributes table of a ClassFile (§4.1) structure. If the constant pool of a class or interface \(C\) con-

---

\(^6\) The InnerClasses attribute was introduced in JDK release 1.1 to support nested classes and interfaces.
contains a CONSTANT_Class_info entry which represents a class or interface that is not a member of a package, then C's ClassFile structure must have exactly one InnerClasses attribute in its attributes table.

The InnerClasses attribute has the following format:

```
InnerClasses_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    { u2 inner_class_info_index;
        u2 outer_class_info_index;
        u2 inner_name_index;
        u2 inner_class_access_flags;
    } classes[number_of_classes];
}
```

The items of the InnerClasses_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 (§4.4.7) structure representing the string "InnerClasses".

- **attribute_length**
  The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

- **number_of_classes**
  The value of the number_of_classes item indicates the number of entries in the classes array.

- **classes[]**
  Every CONSTANT_Class_info entry in the constant_pool table which represents a class or interface C that is not a package member must have exactly one corresponding entry in the classes array.

  If a class has members that are classes or interfaces, its constant_pool table (and hence its InnerClasses attribute) must refer to each such member, even if that member is not otherwise mentioned by the class. These rules imply that a nested
class or interface member will have InnerClasses information for each enclosing class and for each immediate member.

Each classes array entry contains the following four items:

inner_class_info_index
The value of the inner_class_info_index item must be zero or a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing C. The remaining items in the classes array entry give information about C.

outer_class_info_index
If C is not a member, the value of the outer_class_info_index item must be zero. Otherwise, the value of the outer_class_info_index item must be a valid index into the constant_pool table, and the entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing the class or interface of which C is a member.

inner_name_index
If C is anonymous, the value of the inner_name_index item must be zero. Otherwise, the value of the inner_name_index item must be a valid index into the constant_pool table, and the entry at that index must be a CONSTANT_Utf8_info (§4.4.7) structure that represents the original simple name of C, as given in the source code from which this class file was compiled.

inner_class_access_flags
The value of the inner_class_access_flags item is a mask of flags used to denote access permissions to and properties of class or interface C as declared in the source code from which this class file was compiled. It is used by compilers to recover the original information when source code is not available. The flags are shown in Table 4.8.
Table 4.8  Nested class access and property flags

<table>
<thead>
<tr>
<th>Flag Name</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_PUBLIC</td>
<td>0x0001</td>
<td>Marked or implicitly public in source.</td>
</tr>
<tr>
<td>ACC_PRIVATE</td>
<td>0x0002</td>
<td>Marked private in source.</td>
</tr>
<tr>
<td>ACC_PROTECTED</td>
<td>0x0004</td>
<td>Marked protected in source.</td>
</tr>
<tr>
<td>ACC_STATIC</td>
<td>0x0008</td>
<td>Marked or implicitly static in source.</td>
</tr>
<tr>
<td>ACC_FINAL</td>
<td>0x0010</td>
<td>Marked final in source.</td>
</tr>
<tr>
<td>ACC_INTERFACE</td>
<td>0x0200</td>
<td>Was an interface in source.</td>
</tr>
<tr>
<td>ACC_ABSTRACT</td>
<td>0x0400</td>
<td>Marked or implicitly abstract in source.</td>
</tr>
<tr>
<td>ACC_SYNTHETIC</td>
<td>0x1000</td>
<td>Declared synthetic; not present in the source code.</td>
</tr>
<tr>
<td>ACC_ANNOTATION</td>
<td>0x2000</td>
<td>Declared as an annotation type.</td>
</tr>
<tr>
<td>ACC_ENUM</td>
<td>0x4000</td>
<td>Declared as an enum type.</td>
</tr>
</tbody>
</table>

All bits of the inner_class_access_flags item not assigned in Table 4.8 are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

Sun’s Java virtual machine implementation does not check the consistency of an InnerClasses attribute against a class file representing a class or interface referenced by the attribute.

4.7.7  The EnclosingMethod Attribute

The EnclosingMethod attribute is an optional fixed-length attribute in the attributes table of a ClassFile (§4.1) structure. A class must have an EnclosingMethod attribute if and only if it is a local class or an anonymous class. A class may have no more than one EnclosingMethod attribute.

The EnclosingMethod attribute has the following format:
EnclosingMethod_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 class_index
    u2 method_index;
}

The items of the EnclosingMethod_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 (§4.4.7) structure representing the string "EnclosingMethod".

attribute_length
   The value of the attribute_length item is four.

class_index
   The value of the class_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info (§4.4.1) structure representing the innermost class that encloses the declaration of the current class.

method_index
   If the current class is not immediately enclosed by a method or constructor, then the value of the method_index item must be zero. Otherwise, the value of the method_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_NameAndType_info (§4.4.6) structure representing a the name and type of a method in the class referenced by the class_index attribute above. It is the responsibility of the Java compiler to ensure that the method identified via the method_index is indeed the closest lexically enclosing method of the class that contains this EnclosingMethod attribute.
4.7.8 The Synthetic Attribute

The Synthetic attribute\(^7\) is a fixed-length attribute in the attributes table of a ClassFile (§4.1), field_info (§4.5) or method_info (§4.6) structure. A class member that does not appear in the source code must be marked using a Synthetic attribute, or else it must have its ACC_SYNTHETIC flag set. The only exceptions to this requirement are compiler-generated methods which are not considered implementation artifacts, namely the instance initialization method representing a default constructor of the Java programming language (§3.9), the class initialization method (§3.9) and the Enum.values() and Enum.valueOf() methods.

The Synthetic attribute has the following format:

\[
\text{Synthetic\_attribute} \{ \\
  \text{u2 attribute\_name\_index; } \\
  \text{u4 attribute\_length; } \\
\}
\]

The items of the Synthetic\_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 (§4.4.7) structure representing the string "Synthetic".

- **attribute\_length**
  - The value of the attribute\_length item is zero.

4.7.9 The Signature Attribute

The Signature attribute is an optional fixed-length attribute in the attributes table of a ClassFile (§4.1), field\_info (§4.5) or method\_info (§4.6) structure. The Signature attribute records generic signature information for any class, interface, constructor or member whose generic signature in the Java programming language would include references to type variables or parameterized types.

The Signature attribute has the following format:

\(^7\) The Synthetic attribute was introduced in JDK release 1.1 to support nested classes and interfaces.
Signature_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 signature_index;
}

The items of the Signature_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 (§4.4.7) structure representing the string "Signature".

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

signature_index
    The value of the signature_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 (§4.4.7) structure representing either a class signature, if this signature attribute is an attribute of a ClassFile structure, a method type signature, if this signature is an attribute of a method_info structure, or a field type signature otherwise.
4.7.10 The SourceFile Attribute

The SourceFile attribute is an optional fixed-length attribute in the attributes table of a ClassFile (§4.1) structure. There can be no more than one SourceFile attribute in the attributes table of a given ClassFile structure.

The SourceFile attribute has the following format:

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

The items of the SourceFile_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 (§4.4.7) structure representing the string "SourceFile".

attribute_length

- The value of the attribute_length item of a SourceFile_attribute structure must be 2.

sourcefile_index

- The value of the sourcefile_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 (§4.4.7) structure representing a string.

  The string referenced by the sourcefile_index item will be interpreted as indicating the name of the source file from which this class file was compiled. It will not be interpreted as indicating the name of a directory containing the file or an absolute path name for the file; such platform-specific additional information must be supplied by the runtime interpreter or development tool at the time the file name is actually used.
4.7.11 The SourceDebugExtension Attribute

The SourceDebugExtension attribute is an optional attribute in the attributes table of a ClassFile (§4.1) structure. There can be no more than one SourceDebugExtension attribute in the attributes table of a given ClassFile structure.

The SourceDebugExtension attribute has the following format:

```
SourceDebugExtension_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 debug_extension[attribute_length];
}
```

The items of the SourceDebugExtension_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 (§4.4.7) structure representing the string "SourceDebugExtension".

attribute_length

The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes. The value of the attribute_length item is thus the number of bytes in the debug_extension[] item.

d debug_extension[]

The debug_extension array holds a string, which must be in UTF-8 format. There is no terminating zero byte. The string in the debug_extension item will be interpreted as extended debugging information. The content of this string has no semantic effect on the Java Virtual Machine.

4.7.12 The LineNumberTable Attribute

The LineNumberTable attribute is an optional variable-length attribute in the attributes table of a Code (§4.7.3) attribute. It may be used by debuggers to determine which part of the Java virtual machine code array corresponds to a given line number in the original source file. If LineNumberTable attributes are present in the attributes table of a given Code attribute, then they may appear in any
order. Furthermore, multiple LineNumberTable attributes may together represent a given line of a source file; that is, LineNumberTable attributes need not be one-to-one with source lines.

The LineNumberTable attribute has the following format:

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    { u2 start_pc;
      u2 line_number;
    } line_number_table[line_number_table_length];
}
```

The items of the LineNumberTable_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 (§4.4.7) structure representing the string "LineNumberTable".

attribute_length
The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

line_number_table_length
The value of the line_number_table_length item indicates the number of entries in the line_number_table array.

line_number_table[]
Each entry in the line_number_table array indicates that the line number in the original source file changes at a given point in the code array. Each line_number_table entry must contain the following two items:

start_pc
The value of the start_pc item must indicate the index into the code array at which the code for a new line in the original source file begins. The value of start_pc must be less than
the value of the code_length item of the Code attribute of which this LineNumberTable is an attribute.

line_number
The value of the line_number item must give the corresponding line number in the original source file.

4.7.13 The LocalVariableTable Attribute

The LocalVariableTable attribute is an optional variable-length attribute in a Code (§4.7.3) attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method. If LocalVariableTable attributes are present in the attributes table of a given Code attribute, then they may appear in any order. There may be no more than one LocalVariableTable attribute per local variable in the Code attribute.

The LocalVariableTable attribute has the following format:

LocalVariableTable_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 local_variable_table_length;
  {  u2 start_pc;
      u2 length;
      u2 name_index;
      u2 descriptor_index;
      u2 index;
  } local_variable_table[local_variable_table_length];
}

The items of the LocalVariableTable_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 (§4.4.7) structure representing the string "LocalVariableTable".

attribute_length
The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.
The value of the `local_variable_table_length` item indicates the number of entries in the `local_variable_table` array.

Each entry in the `local_variable_table` array indicates a range of code array offsets within which a local variable has a value. It also indicates the index into the local variable array of the current frame at which that local variable can be found. Each entry must contain the following five items:

- **start_pc, length**
  The given local variable must have a value at indices into the code array in the interval `[start_pc, start_pc+length)`, that is, between `start_pc` and `start_pc+length` exclusive. The value of `start_pc` must be a valid index into the code array of this `Code` attribute and must be the index of the opcode of an instruction. The value of `start_pc+length` must either be a valid index into the code array of this `Code` attribute and be the index of the opcode of an instruction, or it must be the first index beyond the end of that code array.

- **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 contain a `CONSTANT_Utf8_info` (§4.4.7) structure representing a valid unqualified name (§4.2.2) denoting a local variable.

- **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 contain a `CONSTANT_Utf8_info` (§4.4.7) structure. That `CONSTANT_Utf8_info` structure must represent a field descriptor (§4.3.2) encoding the type of a local variable in the source program.
The given local variable must be at index in the local variable array of the current frame. If the local variable at index is of type double or long, it occupies both index and index+1.

4.7.14 The LocalVariableTypeTable Attribute

The LocalVariableTypeTable attribute is an optional variable-length attribute in a Code (§4.7.3) attribute. It may be used by debuggers to determine the value of a given local variable during the execution of a method. If LocalVariableTypeTable attributes are present in the attributes table of a given Code attribute, then they may appear in any order. There may be no more than one LocalVariableTypeTable attribute per local variable in the Code attribute.

The LocalVariableTypeTable attribute differs from the LocalVariableTable attribute in that it provides signature information rather than descriptor information. This difference is only significant for variables whose type is a generic reference type. Such variables will appear in both tables, while variables of other types will appear only in LocalVariableTable.

The LocalVariableTypeTable attribute has the following format:

```
LocalVariableTypeTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_type_table_length;
    { u2 start_pc;
        u2 length;
        u2 name_index;
        u2 signature_index;
        u2 index;
    } local_variable_type_table[local_variable_type_table_length];
}
```

The items of the LocalVariableTypeTable_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 (§4.4.7) structure representing the string "LocalVariableTypeTable".

attribute_length
The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes.

local_variable_type_table_length
The value of the local_variable_type_table_length item indicates the number of entries in the local_variable_table array.

local_variable_type_table[]
Each entry in the local_variable_type_table array indicates a range of code array offsets within which a local variable has a value. It also indicates the index into the local variable array of the current frame at which that local variable can be found. Each entry must contain the following five items:

start_pc, length
The given local variable must have a value at indices into the code array in the interval [start_pc, start_pc+length), that is, between start_pc and start_pc+length exclusive. The value of start_pc must be a valid index into the code array of this Code attribute and must be the index of the opcode of an instruction. The value of start_pc+length must either be a valid index into the code array of this Code attribute and be the index of the opcode of an instruction, or it must be the first index beyond the end of that code array.

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 contain a CONSTANT_Utf8_info (§4.4.7)
structure representing a valid unqualified name (§4.2.2) denoting a local variable.

**signature_index**

The value of the signature_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must contain a CONSTANT_Utf8_info (§4.4.7) structure representing a field type signature (§4.3.4) encoding the type of a local variable in the source program.

**index**

The given local variable must be at index in the local variable array of the current frame. If the local variable at index is of type double or long, it occupies both index and index+1.

### 4.7.15 The Deprecated Attribute

The Deprecated attribute\(^8\) is an optional fixed-length attribute in the attributes table of a ClassFile (§4.1), field_info (§4.5) or method_info (§4.6) structure. A class, interface, method, or field may be marked using a Deprecated attribute to indicate that the class, interface, method, or field has been superseded. A runtime interpreter or tool that reads the class file format, such as a compiler, can use this marking to advise the user that a superseded class, interface, method, or field is being referred to. The presence of a Deprecated attribute does not alter the semantics of a class or interface.

The Deprecated attribute has the following format:

```
Deprecated_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
}
```

The items of the Deprecated_attribute structure are as follows:

---

\(^8\) The Deprecated attribute was introduced in JDK release 1.1 to support the @deprecated tag in documentation comments.
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 (§4.4.7) structure representing the string "Deprecated".

attribute_length

The value of the attribute_length item is zero.

4.7.16 The RuntimeVisibleAnnotations attribute

The RuntimeVisibleAnnotations attribute is a variable-length attribute in the attributes table of a ClassFile (§4.1), field_info (§4.5) or method_info (§4.6) structure. The RuntimeVisibleAnnotations attribute records runtime-visible Java programming language annotations on the corresponding class, field, or method. Each ClassFile, field_info, and method_info structure may contain at most one RuntimeVisibleAnnotations attribute, which records all the runtime-visible Java programming language annotations on the corresponding program element. The Java virtual machine must make these annotations available so they can be returned by the appropriate reflective APIs.

The RuntimeVisibleAnnotations attribute has the following format:

RuntimeVisibleAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_annotations;
    annotation annotations[num_annotations];
}

The items of the RuntimeVisibleAnnotations 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 structure representing the string "RuntimeVisibleAnnotations".

attribute_length

The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes. The value of the
attribute_length item is thus dependent on the number of runtime-visible annotations represented by the structure, and their values.

num_annotations
The value of the num_annotations item gives the number of runtime-visible annotations represented by the structure. Note that a maximum of 65535 runtime-visible Java programming language annotations may be directly attached to a program element.

annotations
Each value of the annotations table represents a single runtime-visible annotation on a program element. The annotation structure has the following format:

```
annotation {
    u2 type_index;
    u2 num_element_value_pairs;
    { u2 element_name_index;
        element_value value;
    } element_value_pairs[num_element_value_pairs]
}
```

The items of the annotation structure are as follows:

type_index
The value of the type_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 structure representing a field descriptor representing the annotation type corresponding to the annotation represented by this annotation structure.

num_element_value_pairs
The value of the num_element_value_pairs item gives the number of element-value pairs of the annotation represented by this annotation structure. Note that a maximum of 65535 element-value pairs may be contained in a single annotation.

element_value_pairs
Each value of the element_value_pairs table represents a single element-value pair in the annotation represented by this
annotation structure. Each `element_value_pairs` entry contains the following two items:

`element_name_index`

The value of the `element_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` structure representing the name of the annotation type element represented by this `element_value_pairs` entry.

`value`

The value of the `value` item represents the value of the element-value pair represented by this `element_value_pairs` entry.

### 4.7.16.1 The `element_value` structure

The `element_value` structure is a discriminated union representing the value of an element-value pair. It is used to represent element values in all attributes that describe annotations (`RuntimeVisibleAnnotations`, `RuntimeInvisibleAnnotations`, `RuntimeVisibleParameterAnnotations`, and `RuntimeInvisibleParameterAnnotations`).

The `element_value` structure has the following format:

```plaintext
element_value {
    u1 tag;
    union {
        u2 const_value_index;
        { u2 type_name_index;
            u2 const_name_index;
        } enum_const_value;
        u2 class_info_index;
        annotation annotation_value;
        { u2 num_values;
            element_value values[num_values];
        } array_value;
    } value;
}
```

The items of the `element_value` structure are as follows:
**tag**

The tag item indicates the type of this annotation element-value pair. The letters 'B', 'C', 'D', 'F', 'I', 'J', 'S', and 'Z' indicate a primitive type. These letters are interpreted as **BaseType** characters (Table 4.2). The other legal values for tag are listed with their interpretations in the table below.

**value**

The value item represents the value of this annotation element. This item is a union. The tag item, above, determines which item of the union is to be used:

**const_value_index**

The const_value_index item is used if the tag item is one of 'B', 'C', 'D', 'F', 'I', 'J', 'S', 'Z', or 's'. The value of the const_value_index item must be a valid index into the constant_pool table. The constant_pool entry at that index must be of the correct entry type for the field type designated by the tag item, as specified in Table 4.9.

**enum_const_value**

The enum_const_value item is used if the tag item is 'e'. The enum_const_value item consists of the following two items:

**type_name_index**

The value of the type_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 structure representing the internal

---

**Table 4.9 Interpretation of additional tag values**

<table>
<thead>
<tr>
<th>tag value</th>
<th>Element Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>String</td>
</tr>
<tr>
<td>e</td>
<td>enum constant</td>
</tr>
<tr>
<td>c</td>
<td>class</td>
</tr>
<tr>
<td>@</td>
<td>annotation type</td>
</tr>
<tr>
<td>[</td>
<td>array</td>
</tr>
</tbody>
</table>
form of the binary name (§4.2.1) of the type of the enum
constant represented by this element_value structure.

const_name_index
The value of the const_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 structure representing the simple
name of the enum constant represented by this
element_value structure.

class_info_index
The class_info_index item is used if the tag item is 'c'.
The class_info_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 structure representing
the return descriptor (§4.3.3) of the type that is reified by the
class represented by this element_value structure (e.g., 'V'
for Void.class, 'Ljava/lang/Object;' for Object, etc.)

annotation_value
The annotation_value item is used if the tag item is '@'.
The element_value structure represents a "nested"
annotation.

array_value
The array_value item is used if the tag item is '['. The
array_value item consists of the following two items:

num_values
The value of the num_values item gives the number of
elements in the array-typed value represented by this
element_value structure. Note that a maximum of
65535 elements are permitted in an array-typed element
value.

values
Each value of the values table gives the value of an
element of the array-typed value represented by this
element_value structure.
**4.7.17 The RuntimeInvisibleAnnotations attribute**

The RuntimeInvisibleAnnotations attribute is similar to the RuntimeVisibleAnnotations attribute, except that the annotations represented by a RuntimeInvisibleAnnotations attribute must not be made available for return by reflective APIs, unless the Java virtual machine has been instructed to retain these annotations via some implementation-specific mechanism such as a command line flag. In the absence of such instructions, the Java virtual machine ignores this attribute.

The RuntimeInvisibleAnnotations attribute is a variable-length attribute in the attributes table of a ClassFile (§4.1), field_info (§4.5) or method_info (§4.6) structure. The RuntimeInvisibleAnnotations attribute records runtime-invisible Java programming language annotations on the corresponding class, method, or field. Each ClassFile, field_info, and method_info structure may contain at most one RuntimeInvisibleAnnotations attribute, which records all the runtime-invisible Java programming language annotations on the corresponding program element.

The RuntimeInvisibleAnnotations attribute has the following format:

```java
RuntimeInvisibleAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 num_annotations;
    annotation annotations[num_annotations];
}
```

The items of the RuntimeInvisibleAnnotations 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 structure representing the string "RuntimeInvisibleAnnotations".

**attribute_length**

The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes. The value of the attribute_length item is thus dependent on the number of
runtime-invisible annotations represented by the structure, and
their values.

num_annotations
The value of the num_annotations item gives the number of
runtime-invisible annotations represented by the structure. Note
that a maximum of 65535 runtime-invisible Java programming
language annotations may be directly attached to a program
element.

annotations
Each value of the annotations table represents a single runtime-
invisible annotation on a program element.

4.7.18 The RuntimeVisibleParameterAnnotations attribute

The RuntimeVisibleParameterAnnotations attribute is a variable-length
attribute in the attributes table of the method_info (§4.6) structure. The Run-
timeVisibleParameterAnnotations attribute records runtime-visible Java pro-
gramming language annotations on the parameters of the corresponding method.
Each method_info structure may contain at most one RuntimeVisiblePara-
meterAnnotations attribute, which records all the runtime-visible Java program-
ing language annotations on the parameters of the corresponding method. The
Java virtual machine must make these annotations available so they can be returned
by the appropriate reflective APIs.

The RuntimeVisibleParameterAnnotations attribute has the following
format:

RuntimeVisibleParameterAnnotations_attribute {
   u2 attribute_name_index;
   u4 attribute_length;
   u1 num_parameters;
   {    u2 num_annotations;
        annotation annotations[num_annotations];
   }    parameter_annotations[num_parameters];
}

The items of the RuntimeVisibleParameterAnnotations structure are as fol-
lows:
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 structure representing the string "RuntimeVisibleParameterAnnotations".

attribute_length
The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes. The value of the attribute_length item is thus dependent on the number of parameters, the number of runtime-visible annotations on each parameter, and their values.

num_parameters
The value of the num_parameters item gives the number of parameters of the method represented by the method_info structure on which the annotation occurs. (This duplicates information that could be extracted from the method descriptor.)

parameter_annotations
Each value of the parameter_annotations table represents all of the runtime-visible annotations on a single parameter. The sequence of values in the table corresponds to the sequence of parameters in the method descriptor.
Each parameter_annotations entry contains the following two items:

num_annotations
The value of the num_annotations item indicates the number of runtime-visible annotations on the parameter corresponding to the sequence number of this parameter_annotations element.

annotations
Each value of the annotations table represents a single runtime-visible annotation on the parameter corresponding to the sequence number of this parameter_annotations element.
4.7.19 The RuntimeInvisibleParameterAnnotations attribute

The RuntimeInvisibleParameterAnnotations attribute is similar to the RuntimeVisibleParameterAnnotations attribute, except that the annotations represented by a RuntimeInvisibleParameterAnnotations attribute must not be made available for return by reflective APIs, unless the Java virtual machine has specifically been instructed to retain these annotations via some implementation-specific mechanism such as a command line flag. In the absence of such instructions, the Java virtual machine ignores this attribute.

The RuntimeInvisibleParameterAnnotations attribute is a variable-length attribute in the attributes table of a method_info (§4.6) structure. The RuntimeInvisibleParameterAnnotations attribute records runtime-invisible Java programming language annotations on the parameters of the corresponding method. Each method_info structure may contain at most one RuntimeInvisibleParameterAnnotations attribute, which records all the runtime-invisible Java programming language annotations on the parameters of the corresponding method.

The RuntimeInvisibleParameterAnnotations attribute has the following format:

```java
RuntimeInvisibleParameterAnnotations_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 num_parameters;
    {   u2 num_annotations;
        annotation annotations[num_annotations];
    }   parameter_annotations[num_parameters];
}
```

The items of the RuntimeInvisibleParameterAnnotations 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 structure representing the string "RuntimeInvisibleParameterAnnotations".
attribute_length

The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes. The value of the attribute_length item is thus dependent on the number of parameters, the number of runtime-invisible annotations on each parameter, and their values.

num_parameters

The value of the num_parameters item gives the number of parameters of the method represented by the method_info structure on which the annotation occurs. (This duplicates information that could be extracted from the method descriptor.)

parameter_annotations

Each value of the parameter_annotations table represents all of the runtime-invisible annotations on a single parameter. The sequence of values in the table corresponds to the sequence of parameters in the method descriptor. Each parameter_annotations entry contains the following two items:

num_annotations

The value of the num_annotations item indicates the number of runtime-invisible annotations on the parameter corresponding to the sequence number of this parameter_annotations element.

annotations

Each value of the annotations table represents a single runtime-invisible annotation on the parameter corresponding to the sequence number of this parameter_annotations element.

4.7.20 The AnnotationDefault attribute

The AnnotationDefault attribute is a variable-length attribute in the attributes table of certain method_info (§4.6) structures, namely those representing elements of annotation types. The AnnotationDefault attribute records the default value for the element represented by the method_info structure. Each method_info structures representing an element of an annotation types may contain at most one
AnnotationDefault attribute. The Java virtual machine must make this default value available so it can be applied by appropriate reflective APIs.

The AnnotationDefault attribute has the following format:

```java
AnnotationDefault_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    element_value default_value;
}
```

The items of the AnnotationDefault 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 structure representing the string "AnnotationDefault".

attribute_length
The value of the attribute_length item indicates the length of the attribute, excluding the initial six bytes. The value of the attribute_length item is thus dependent on the default value.

default_value
The default_value item represents the default value of the annotation type element whose default value is represented by this AnnotationDefault attribute.


4.8 Format Checking

When a prospective class file is loaded (§5.3) by the Java virtual machine, the Java virtual machine first ensures that the file has the basic format of a class file (§4.1). This process is known as format checking. The first four bytes must contain the right magic number. All recognized attributes must be of the proper length. The class file must not be truncated or have extra bytes at the end. The constant pool must not contain any superficially unrecognizable information.

This check for basic class file integrity is necessary for any interpretation of the class file contents.

Format checking is distinct from bytecode verification. Both are part of the verification process. Historically, format checking has been confused with bytecode verification, because both are a form of integrity check.

4.9 Constraints on Java Virtual Machine Code

The Java virtual machine code for a method, instance initialization method (§3.9), or class or interface initialization method (§3.9) is stored in the code array of the Code attribute of a method_info structure of a class file. This section describes the constraints associated with the contents of the Code_attribute structure.

4.9.1 Static Constraints

The static constraints on a class file are those defining the well-formedness of the file. With the exception of the static constraints on the Java virtual machine code of the class file, these constraints have been given in the previous sections. The static constraints on the Java virtual machine code in a class file specify how Java virtual machine instructions must be laid out in the code array and what the operands of individual instructions must be.

The static constraints on the instructions in the code array are as follows:

- The code array must not be empty, so the code_length item cannot have the value 0.
- The value of the code_length item must be less than 65536.
• The opcode of the first instruction in the code array begins at index 0.

• Only instances of the instructions documented in Section 6.4 may appear in the code array. Instances of instructions using the reserved opcodes (§6.2) or any opcodes not documented in this specification must not appear in the code array.

• If the class file version number is 51.0 or above, then neither the jsr opcode or the jsr_w opcode may appear in the code array.

• For each instruction in the code array except the last, the index of the opcode of the next instruction equals the index of the opcode of the current instruction plus the length of that instruction, including all its operands. The wide instruction is treated like any other instruction for these purposes; the opcode specifying the operation that a wide instruction is to modify is treated as one of the operands of that wide instruction. That opcode must never be directly reachable by the computation.

• The last byte of the last instruction in the code array must be the byte at index code_length–1.

The static constraints on the operands of instructions in the code array are as follows:

• The target of each jump and branch instruction (jsr, jsr_w, goto, goto_w, ifeq, ifne, ifle, iflt, ifge, ifgt, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmple, if_icmplt, if_icmpge, if_icmpgt, if_acmpeq, if_acmpne) must be the opcode of an instruction within this method. The target of a jump or branch instruction must never be the opcode used to specify the operation to be modified by a wide instruction; a jump or branch target may be the wide instruction itself.

• Each target, including the default, of each tableswitch instruction must be the opcode of an instruction within this method. Each tableswitch instruction must have a number of entries in its jump table that is consistent with the value of its low and high jump table operands, and its low value must be less than or equal to its high value. No target of a tableswitch instruction may be the opcode used to specify the operation to be modified by a wide instruction; a tableswitch target may be a wide instruction itself.

• Each target, including the default, of each lookupswitch instruction must be the opcode of an instruction within this method. Each lookupswitch instruction
must have a number of `match-offset` pairs that is consistent with the value of its `npairs` operand. The `match-offset` pairs must be sorted in increasing numerical order by signed `match` value. No target of a `lookupswitch` instruction may be the opcode used to specify the operation to be modified by a `wide` instruction; a `lookupswitch` target may be a `wide` instruction itself.

- The operand of each `ldc` instruction must be a valid index into the `constant_pool` table. The operands of each `ldc_w` instruction must represent a valid index into the `constant_pool` table. In both cases, the constant pool entry referenced by that index must be of type:
  - `CONSTANT_Integer`, `CONSTANT_Float` or `CONSTANT_String` if the class file version number is less than 49.0.
  - `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_String` or `CONSTANT_Class` if the class file version number is 49.0 or above.

- The operands of each `ldc2_w` instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Long` or `CONSTANT_Double`. In addition, the subsequent constant pool index must also be a valid index into the constant pool, and the constant pool entry at that index must not be used.

- The operands of each `getfield`, `putfield`, `getstatic`, and `putstatic` instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Fieldref`.

- The indexbyte operands of each `invokevirtual`, `invokespecial`, and `invokestatic` instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_Methodref`.

- Only the `invokespecial` instruction is allowed to invoke an instance initialization method (§3.9). No other method whose name begins with the character `'<` (`'\u003c'`) may be called by the method invocation instructions. In particular, the class or interface initialization method specially named `<clinit>` is never called explicitly from Java virtual machine instructions, but only implicitly by the Java virtual machine itself.

- The indexbyte operands of each `invokeinterface` instruction must represent a valid index into the `constant_pool` table. The constant pool entry referenced by that index must be of type `CONSTANT_InterfaceMethodref`. The value of the `count` operand of each `invokeinterface`
instruction must reflect the number of local variables necessary to store
the arguments to be passed to the interface method, as implied by the
descriptor of the CONSTANT_NameAndType_info structure referenced by the
CONSTANT_InterfaceMethodref constant pool entry. The fourth operand
byte of each invokeinterface instruction must have the value zero.

• The operands of each instanceof, checkcast, new, and anewarray instruction
and the indexbyte operands of each multianewarray instruction must repre-
sent a valid index into the constant_pool table. The constant pool entry
referenced by that index must be of type CONSTANT_Class.

• No anewarray instruction may be used to create an array of more than 255
dimensions.

• No new instruction may reference a CONSTANT_Class constant_pool table
entry representing an array class. The new instruction cannot be used to create
an array.

• A multianewarray instruction must be used only to create an array of a type
that has at least as many dimensions as the value of its dimensions operand.
That is, while a multianewarray instruction is not required to create all of the
dimensions of the array type referenced by its indexbyte operands, it must not
attempt to create more dimensions than are in the array type. The dimensions
operand of each multianewarray instruction must not be zero.

• The atype operand of each newarray instruction must take one of the values
T_BOOLEAN (4), T_CHAR (5), T_FLOAT (6), T_DOUBLE (7), T_BYTE (8), T_SHORT
(9), T_INT (10), or T_LONG (11).

• The index operand of each iload, fload, aload, istore, fstore, astore, iinc, and ret
instruction must be a nonnegative integer no greater than max_locals − 1.

• The implicit index of each iload_<n>, fload_<n>, aload_<n>, istore_<n>,
fstore_<n>, and astore_<n> instruction must be no greater than the value of
max_locals − 1.

• The index operand of each lload, dload, lstore, and dstore instruction must be
no greater than the value of max_locals − 2.

• The implicit index of each lload_<n>, dload_<n>, lstore_<n>, and
dstore_<n> instruction must be no greater than the value of max_locals − 2.

• The indexbyte operands of each wide instruction modifying an iload, fload,
aload, astore, fstore, astore, ret, or iinc instruction must represent a nonnegative
integer no greater than max_locals − 1. The indexbyte operands of each wide instruction modifying an lload, dload, lstore, or dstore instruction must represent a nonnegative integer no greater than max_locals − 2.

4.9.2 Structural Constraints

The structural constraints on the code array specify constraints on relationships between Java virtual machine instructions. The structural constraints are as follows:

- Each instruction must only be executed with the appropriate type and number of arguments in the operand stack and local variable array, regardless of the execution path that leads to its invocation. An instruction operating on values of type int is also permitted to operate on values of type boolean, byte, char, and short. (As noted in §3.3.4 and §3.11.1, the Java virtual machine internally converts values of types boolean, byte, char, and short to type int.)

- If an instruction can be executed along several different execution paths, the operand stack must have the same depth (§3.6.2) prior to the execution of the instruction, regardless of the path taken.

- At no point during execution can the order of the local variable pair holding a value of type long or double be reversed or the pair split up. At no point can the local variables of such a pair be operated on individually.

- No local variable (or local variable pair, in the case of a value of type long or double) can be accessed before it is assigned a value.

- At no point during execution can the operand stack grow to a depth (§3.6.2) greater than that implied by the max_stack item.

- At no point during execution can more values be popped from the operand stack than it contains.

- Each invokespecial instruction must name an instance initialization method (§3.9), a method in the current class, or a method in a superclass of the current class.

- When the instance initialization method (§3.9) is invoked, an uninitialized class instance must be in an appropriate position on the operand stack. An instance initialization method must never be invoked on an initialized class instance.
• When any instance method is invoked or when any instance variable is accessed, the class instance that contains the instance method or instance variable must already be initialized.

• There must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken.

• There must never be an uninitialized class instance on the operand stack or in a local variable when a \texttt{jsr} or \texttt{jsr_w} instruction is executed.

• Each instance initialization method (§3.9), except for the instance initialization method derived from the constructor of class \texttt{Object}, must call either another instance initialization method of \texttt{this} or an instance initialization method of its direct superclass \texttt{super} before its instance members are accessed. However, instance fields of \texttt{this} that are declared in the current class may be assigned before calling any instance initialization method.

• The arguments to each method invocation must be method invocation compatible (JLS3 §5.3) with the method descriptor (§4.3.3).

• The type of every class instance that is the target of a method invocation instruction must be assignment compatible (JLS3 §5.2) with the class or interface type specified in the instruction. In addition, the type of the target of an \texttt{invokespecial} instruction must be assignment compatible with the current class, unless an instance initialization method is being invoked.

• Each return instruction must match its method’s return type. If the method returns a \texttt{boolean}, \texttt{byte}, \texttt{char}, \texttt{short}, or \texttt{int}, only the \texttt{ireturn} instruction may be used. If the method returns a \texttt{float}, \texttt{long}, or \texttt{double}, only an \texttt{freturn}, \texttt{lreturn}, or \texttt{dreturn} instruction, respectively, may be used. If the method returns a \texttt{reference} type, it must do so using an \texttt{areturn} instruction, and the type of the returned value must be assignment compatible (JLS3 §5.2) with the return descriptor (§4.3.3) of the method. All instance initialization methods, class or interface initialization methods, and methods declared to return \texttt{void} must use only the \texttt{return} instruction.

• If \texttt{getfield} or \texttt{putfield} is used to access a \texttt{protected} field of a superclass that is a member of a different runtime package than the current class, then the type of the class instance being accessed must be the same as or a subclass of the current class. If \texttt{invokevirtual} or \texttt{invokespecial} is used to access a \texttt{protected} method of a superclass that is a member of a different runtime
package than the current class, then the type of the class instance being accessed must be the same as or a subclass of the current class.

- The type of every class instance accessed by a `getfield` instruction or modified by a `putfield` instruction must be assignment compatible (JLS3 §5.2) with the class type specified in the instruction.

- The type of every value stored by a `putfield` or `putstatic` instruction must be compatible with the descriptor of the field (§4.3.2) of the class instance or class being stored into. If the descriptor type is `boolean`, `byte`, `char`, `short`, or `int`, then the value must be an `int`. If the descriptor type is `float`, `long`, or `double`, then the value must be a `float`, `long`, or `double`, respectively. If the descriptor type is a `reference` type, then the value must be of a type that is assignment compatible (JLS3 §5.2) with the descriptor type.

- The type of every value stored into an array by an `aastore` instruction must be a `reference` type. The component type of the array being stored into by the `aastore` instruction must also be a `reference` type.

- Each `atthrow` instruction must throw only values that are instances of class `Throwable` or of subclasses of `Throwable`. Each class mentioned in a `catch_type` item of a method’s exception table must be `Throwable` or of subclasses of `Throwable`.

- Execution never falls off the bottom of the code array.

- No return address (a value of type `returnAddress`) may be loaded from a local variable.

- The instruction following each `jsr` or `jsr_w` instruction may be returned to only by a single `ret` instruction.

- No `jsr` or `jsr_w` instruction may be used to recursively call a subroutine if that subroutine is already present in the subroutine call chain. (Subroutines can be nested when using `try-finally` constructs from within a `finally` clause.)

- Each instance of type `returnAddress` can be returned to at most once. If a `ret` instruction returns to a point in the subroutine call chain above the `ret` instruction corresponding to a given instance of type `returnAddress`, then that instance can never be used as a return address.
4.10 Verification of class Files

Even though any compiler for the Java programming language must only produce class files that satisfy all the static and structural constraints in the previous sections, the Java virtual machine has no guarantee that any file it is asked to load was generated by that compiler or is properly formed. Applications such as web browsers do not download source code, which they then compile; these applications download already-compiled class files. The browser needs to determine whether the class file was produced by a trustworthy compiler or by an adversary attempting to exploit the virtual machine.

An additional problem with compile-time checking is version skew. A user may have successfully compiled a class, say PurchaseStockOptions, to be a subclass of TradingClass. But the definition of TradingClass might have changed since the time the class was compiled in a way that is not compatible with preexisting binaries. Methods might have been deleted or had their return types or modifiers changed. Fields might have changed types or changed from instance variables to class variables. The access modifiers of a method or variable may have changed from public to private. For a discussion of these issues, see Chapter 13, “Binary Compatibility,” in the The Java™ Language Specification.

Because of these potential problems, the Java virtual machine needs to verify for itself that the desired constraints are satisfied by the class files it attempts to incorporate. A Java virtual machine implementation verifies that each class file satisfies the necessary constraints at linking time (§5.4).

Linking-time verification enhances the performance of the interpreter. Expensive checks that would otherwise have to be performed to verify constraints at runtime for each interpreted instruction can be eliminated. The Java virtual machine can assume that these checks have already been performed. For example, the Java virtual machine will already know the following:

- There are no operand stack overflows or underflows.
- All local variable uses and stores are valid.
- The arguments to all the Java virtual machine instructions are of valid types.
The verifier also performs verification that can be done without looking at the code array of the Code attribute (§4.7.3). The checks performed include the following:

- Ensuring that final classes are not subclassed and that final methods are not overridden.
- Checking that every class (except Object) has a direct superclass.
- Ensuring that the constant pool satisfies the documented static constraints: for example, that each CONSTANT_Class_info structure in the constant pool contains in its name_index item a valid constant pool index for a CONSTANT_Utf8_info structure.
- Checking that all field references and method references in the constant pool have valid names, valid classes, and a valid type descriptor.

Note that these checks do not ensure that the given field or method actually exists in the given class, nor do they check that the type descriptors given refer to real classes. They ensure only that these items are well formed. More detailed checking is performed when the bytecodes themselves are verified, and during resolution.

There are two strategies that Java virtual machines may use for verification. Verification by type checking must be used to verify class files whose version number is greater than or equal to 50.0.

Verification by type inference must be supported by all Java virtual machines, except those conforming to the Java ME CLDC and Java Card profiles, in order to verify class files whose version number is less than 50.0. Verification on virtual machines supporting the Java ME CLDC and Java Card profiles is governed by their respective specifications.

4.10.1 Verification by Type Checking

A class file whose version number is greater than or equal to 50.0 must be verified using the typechecking rules given in this section. If, and only if, a class file’s version number equals 50.0, then if the typechecking fails, a virtual machine implementation may choose to attempt to perform verification by type inference.
alter the bytecodes of a method in a manner that requires adjustment of
the method’s stack map frames. If a tool does not make the necessary
adjustments to the stack map frames, typechecking may fail even though
the bytecode is in principle valid (and would consequently verify under
the old type inference scheme). To allow implementors time to adapt
their tools, virtual machines may fall back to the older verification
discipline, but only for a limited time.

In cases where typechecking fails but type inference is invoked and
succeeds, a certain performance penalty is expected. Such a penalty is
unavoidable. It also should serve as a signal to tool vendors that their
output needs to be adjusted, and provides vendors with additional
incentive to make these adjustments.

If a virtual machine implementation ever attempts to perform verification by
type inference on version 50.0 class files, it must do so in all cases where
verification by typechecking fails.

This means that that a virtual machine cannot choose to resort to type
inference in once case and not in another. It must either reject class files
that do not verify via typechecking, or else consistently failover to the
type inferencing verifier whenever typechecking fails.

The type checker requires a list of stack map frames for each method with a
Code attribute. The type checker reads the stack map frames for each such method
and uses these maps to generate a proof of the type safety of the instructions in the
Code attribute. The list of stack map frames is given by the StackMapTable
(§4.7.4) attribute of the Code attribute.

The intent is that a stack map frame must appear at the beginning of
each basic block in a method. The stack map frame specifies the
verification type of each operand stack entry and of each local variable
at the start of each basic block.

The type rules that the typechecker enforces are specified by means of Prolog
clauses. English language text is used to describe the type rules in an informal
way, while the Prolog code provides a formal specification.

Iff the predicate classIsTypeSafe is not true, the type checker must throw the
exception VerifyError to indicate that the class file is malformed. Otherwise,
the class file has type checked successfully and bytecode verification has completed successfully.

\[
\text{classIsTypeSafe(Class) :-}
\]
\[
\text{classClassName(Class, Name),}
\]
\[
\text{classDefiningLoader(Class, L),}
\]
\[
\text{superclassChain(Name, L, Chain),}
\]
\[
\text{Chain \not= [],}
\]
\[
\text{classSuperClassName(Class, SuperclassName),}
\]
\[
\text{loadedClass(SuperclassName, L, Superclass),}
\]
\[
\text{classListNotFinal(Superclass),}
\]
\[
\text{classListMethods(Class, Methods),}
\]
\[
\text{checklist(methodIsTypeSafe(Class), Methods).}
\]

\[
\text{classIsTypeSafe(Class) :-}
\]
\[
\text{classClassName(Class, ‘java/lang/Object’),}
\]
\[
\text{classDefiningLoader(Class, L),}
\]
\[
\text{isBootstrapClassLoader(L),}
\]
\[
\text{classListMethods(Class, Methods),}
\]
\[
\text{checklist(methodIsTypeSafe(Class), Methods).}
\]

Thus, a class is type safe if all its methods are type safe, and it does not subclass a final class.

The predicate \text{classListIsTypeSafe} assumes that \text{Class} is a Prolog term representing a binary class that has been successfully parsed and loaded. This specification does not mandate the precise structure of this term, but does require that certain predicates (e.g., \text{classListMethods}) be defined upon it, as specified in Section 4.10.1.3.1.

For example, we assume a predicate \text{classListMethods(Class, Methods)} that, given a term representing a class as described above as its first argument, binds its second argument to a list
comprising all the methods of the class, represented in a convenient form described below.

We also require the existence of a predicate loadedClass(Name, InitiatingLoader, ClassDefinition) which asserts that there exists a class named Name whose representation (in accordance with this specification) when loaded by the class loader InitiatingLoader is ClassDefinition. Additional required predicates are discussed in §4.10.1.3.1.

Individual instructions are presented as terms whose functor is the name of the instruction and whose arguments are its parsed operands.

For example, an aload instruction is represented as the term aload(N), which includes the index N that is the operand of the instruction.

A few instructions have operands that are constant pool entries representing methods or fields. As specified in §4.3.3, methods are represented by CONSTANT_InterfaceMethodref_info (for interface methods) or CONSTANT_Methodref_info (for other methods) structures in the constant pool. Such structures are represented here as functor applications of the form imethod(MethodClassName, MethodName, MethodDescriptor) (for interface methods) or method (MethodClassName, MethodName, MethodDescriptor) (for other methods), where MethodClassName is the name of the class referenced by the class_index item for the structure, and MethodName and MethodDescriptor correspond to the name and type descriptor referenced by the name_and_type_index item of the structure.

Similarly, fields are represented by CONSTANT_Fieldref_info structures in the class file. These structures are represented here as functor applications of the form field(FieldClassName, FieldName, FieldDescriptor) where FieldClassName is the name of the class referenced by the class_index item in the structure, and FieldName and FieldDescriptor correspond to the name and type descriptor referenced by the name_and_type_index item of the structure. For clarity, we assume that type descriptors are mapped into more readable names: the leading L and trailing ; are dropped from class names, and the base type characters used for primitive types are mapped to the names of those types).
So, a `getfield` instruction whose operand was an index into the constant pool that refers to a field `foo` of type `F` in class `Bar` would be represented as `getfield(field('Bar', 'foo', 'F')).`

Constant pool entries that refer to constant values, such as `CONSTANT_String`, `CONSTANT_Integer`, `CONSTANT_Float`, `CONSTANT_Long`, `CONSTANT_Double` and `CONSTANT_Class`, are encoded via the functors whose names are `string`, `int`, `float`, `long`, `double` and `classConstant` respectively.

So an `ldc` instruction for loading the integer 91 would be encoded as `ldc(int(91))`.

The instructions as a whole are represented as a list of terms of the form `instruction(Offset, AnInstruction)`.

For example, `instruction(21, aload(1))`.

The order of instructions in this list must be the same as in the class file.

Stack map frames are represented as a list of terms of the form `stackMap(Offset, TypeState)` where `Offset` is an integer indicating the offset of the instruction the frame map applies to, and `TypeState` is the expected incoming type state for that instruction. The order of instructions in this list must be the same as in the class file.

`TypeState` has the form `frame(Locals, OperandStack, Flags)`.

`Locals` is a list of verification types, such that the Nth element of the list (with 0 based indexing) represents the type of local variable N. If any local variable in `Locals` has the type `uninitializedThis`, `Flags` is `[flagThisUninit]`, otherwise it is an empty list.

`OperandStack` is a list of types, such that the first element represents the type of the top of the operand stack, and the elements below the top follow in the appropriate order.

However, note again that types of size 2 are represented by two entries, with the first entry being top and the second one being the type itself.
So, a stack with a `double`, an `int` and a `long` would be represented as `[top, double, int, top, long]`.

Array types are represented by applying the functor `arrayOf` to an argument denoting the component type of the array. Other reference types are represented using the functor `class`. Hence `class(N, L)` represents the class whose binary name is `N` as loaded by the loader `L`.

Thus, `L` is an initiating loader of the class represented by `class(N, L)`. It may, or may not, be its defining loader.

The type `uninitialized(offset)` is represented by applying the functor `uninitialized` to an argument representing the numerical value of the `offset`. Other verification types are represented by Prolog atoms whose name denotes the verification type in question.

So, the class `Object` would be represented as `class(java/lang/Object', BL)`, where `BL` is the bootstrap loader. The types `int[]` and `Object[]` would be represented by `arrayOf(int)` and `arrayOf(class('java/lang/Object', BL))` respectively.

Flags is a list which may either be empty or have the single element `flagThisUninit`.

This flag is used in constructors, to mark type states where initialization of `this` has not yet been completed. In such type states, it is illegal to return from the method.

### 4.10.1.1 The Type Hierarchy

The typechecker enforces a type system based upon a hierarchy of verification types, illustrated in figure 1. Most verifier types have a direct correspondence with Java virtual machine field type descriptors as given in Table 4.2. The only exceptions are the field descriptors `B`, `C`, `S` and `Z` all of which correspond to the verifier type `int`.
Figure 1: The verification type Hierarchy
4.10.1.2 Subtyping Rules

Subtyping is reflexive.

\[
isAssignble(X, X) .
\]

\[
isAssignble(\text{oneWord}, \text{top}).
\]

\[
isAssignble(\text{twoWord}, \text{top}).
\]

\[
isAssignble(\text{int}, X) :- \text{isAssignble(\text{oneWord}, X)}.
\]

\[
isAssignble(\text{float}, X) :- \text{isAssignble(\text{oneWord}, X)}.
\]

\[
isAssignble(\text{long}, X) :- \text{isAssignble(\text{twoWord}, X)}.
\]

\[
isAssignble(\text{double}, X) :- \text{isAssignble(\text{twoWord}, X)}.
\]

\[
isAssignble(\text{reference}, X) :- \text{isAssignble(\text{oneWord}, X)}.
\]

\[
isAssignble(\text{class}(\_ , \_ ), X) :- \text{isAssignble(\text{reference}, X)}.
\]

\[
isAssignble(\text{arrayOf}(\_), X) :- \text{isAssignble(\text{reference}, X)}.
\]

\[
isAssignble(\text{uninitialized}, X) :- \text{isAssignble(\text{reference}, X)}.
\]

\[
isAssignble(\text{uninitializedThis}, X) :- \text{isAssignble(\text{uninitialized}, X)}.
\]

\[
isAssignble(\text{uninitialized}(\_), X) :- \text{isAssignble(\text{uninitialized}, X)}.
\]

\[
isAssignble(\text{null}, \text{class}(\_, \_)).
\]

\[
isAssignble(\text{null}, \text{arrayOf}(\_)).
\]

\[
isAssignble(\text{null}, X) :-
\]

\[\begin{align*}
& \quad \text{isAssignble(\text{class}(\text{'java/lang/Object'}, BL), X),} \\
& \quad \text{isBootstrapLoader(BL).}
\end{align*}\]
These subtype rules are not necessarily the most obvious formulation of subtyping. There is a clear split between subtyping rules for reference types in the Java programming language, and rules for the remaining verification types. The split allows us to state general subtyping relations between Java programming language types and other verification types. These relations hold independently of a Java type’s position in the class hierarchy, and help to prevent excessive class loading in a Java virtual machine implementation. For example, we do not want to start climbing up the Java class hierarchy in response to a query of the form \text{class}(\text{foo}, \text{L}) \leq \text{twoWord}.

Subtype rules for the reference types in the Java programming language are specified recursively in the obvious way with \text{isJavaAssignable}. The remaining verification types have subtype rules of the form:

\text{isAssignable}(v, X) \leftarrow \text{isAssignable}(\text{the\_direct\_supertype\_of}\_v, X).

That is, \( v \) is a subtype of \( X \) if the direct supertype of \( v \) is a subtype of \( X \).

We also have a rule that says subtyping is reflexive, so together these rules cover most verification types that are not reference types in the Java programming language.

\begin{verbatim}
isAssignable(class(X, Lx), class(Y, Ly)) :-
    isJavaAssignable(class(X, Lx), class(Y, Ly)).
isAssignable(arrayOf(X), class(Y, L)) :-
    isJavaAssignable(arrayOf(X), class(Y, L)).
isAssignable(arrayOf(X), arrayOf(Y)) :-
    isJavaAssignable(arrayOf(X), arrayOf(Y)).
\end{verbatim}

For assignments, interfaces are treated like Object.

\begin{verbatim}
isJavaAssignable(class(_, _), class(To, L)) :-
    loadedClass(To, L, ToClass),
    classIsInterface(ToClass).
isJavaAssignable(From, To) :-
    isJavaSubclassOf(From, To).
\end{verbatim}
Arrays are subtypes of Object.

isJavaAssignable(arrayOf(_), class('java/lang/Object', BL)) :-
    isBootstrapLoader(BL).

The intent here is that array types are subtypes of Cloneable and java.io.Serializable.

isJavaAssignable(arrayOf(_), X) :-
    isArrayInterface(X).

Subtyping between arrays of primitive type is the identity relation.

isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
    atom(X),
    atom(Y),
    X = Y.

Subtyping between arrays of reference type is covariant.

isJavaAssignable(arrayOf(X), arrayOf(Y)) :-
    compound(X), compound(Y), isJavaAssignable(X, Y).

isArrayInterface(class('java/lang/Cloneable', BL)) :-
    isBootstrapLoader(BL).

isArrayInterface(class('java/io/Serializable', BL)) :-
    isBootstrapLoader(BL).
Subclassing is reflexive.

\[ \text{isJavaSubclassOf}(\text{class}(	ext{SubClassName}, L), \text{class}(	ext{SubClassName}, L)). \]

\[ \text{isJavaSubclassOf}(\text{class}(	ext{SubClassName}, LSub), \text{class}(	ext{SuperClassName}, LSuper)) :- \]
\[ \text{superclassChain}(	ext{SubClassName}, LSub, Chain), \]
\[ \text{member}(	ext{class}(	ext{SuperclassName}, L), Chain), \]
\[ \text{loadedClass}(	ext{SuperClassName}, L, Sup), \]
\[ \text{loadedClass}(	ext{SuperClassName}, LSuper, Sup). \]

\[ \text{sizeOf}(X, 2) :- \text{isAssignable}(X, \text{twoWord}). \]
\[ \text{sizeOf}(X, 1) :- \text{isAssignable}(X, \text{oneWord}). \]
\[ \text{sizeOf}(\text{top}, 1). \]

Subtyping is extended pointwise to type states.

The local variable array of a method has a fixed length by construction (in \text{methodInitialStackFrame}) while the operand stack grows and shrinks. Therefore, we require an explicit check on the length of the operand stacks whose assignability is desired.

\[ \text{frameIsAssignable}(\text{frame}(	ext{Locals1}, \text{StackMap1}, \text{Flags1}), \text{frame}(	ext{Locals2}, \text{StackMap2}, \text{Flags2})) :- \]
\[ \text{length}(	ext{StackMap1}, \text{StackMapLength}), \]
\[ \text{length}(	ext{StackMap2}, \text{StackMapLength}), \]
\[ \text{maplist}(\text{isAssignable}, \text{Locals1}, \text{Locals2}), \]
\[ \text{maplist}(\text{isAssignable}, \text{StackMap1}, \text{StackMap2}), \]
\[ \text{subset}(\text{Flags1}, \text{Flags2}). \]
4.10.1.3 Typechecking Rules

4.10.1.3.1 Accessors

**Stipulated Accessors:** Throughout this specification, we assume the existence of certain Prolog predicates whose formal definitions are not given in the specification. We list these predicates and describe their expected behavior below.

The principle guiding the determination as to which accessors are fully specified and which are stipulated is that we do not want to over-specify the representation of the class file. Providing specific accessors to the class or method term would force us to completely specify a format for the Prolog term representing the class file.

parseFieldDescriptor(Descriptor, Type)

Converts a field descriptor, `Descriptor`, into the corresponding verification type `Type` (see the beginning of Section 4.10.1.1 for the specification of this correspondence).

parseMethodDescriptor(Descriptor, ArgTypeList, ReturnType)

Converts a method descriptor, `Descriptor`, into a list of verification types, `ArgTypeList`, corresponding (Section 4.10.1.1) to the method argument types, and a verification type, `ReturnType`, corresponding to the return type.

parseCodeAttribute(Class, Method, FrameSize, MaxStack, ParsedCode, Handlers, StackMap)

Extracts the instruction stream, `ParsedCode`, of the method `Method` in `Class`, as well as the maximum operand stack size, `MaxStack`, the maximal number of local variables, `FrameSize`, the exception handlers, `Handlers`, and the stack map `StackMap`. The representation of the instruction stream and stack map attribute must be as specified in the beginning of §4.10.1 Each exception handler is represented by a functor application of the form `handler(Start, End, Target, ClassName)` whose arguments are, respectively, the start and end of the range of instructions covered by the handler, the first instruction of the handler code, and the name of the exception class that this handler is designed to handle.
classClassName(Class, ClassName)
   Extracts the name, ClassName, of the class Class.

classIsInterface(Class)
   True iff the class, Class, is an interface.

classIsNotFinal(Class)
   True iff the class, Class, is not a final class.

classSuperClassName(Class, SuperClassName)
   Extracts the name, SuperClassName, of the superclass of class Class.

classInterfaces(Class, Interfaces)
   Extracts a list, Interfaces, of the direct superinterfaces of the class Class.

classMethods(Class, Methods)
   Extracts a list, Methods, of the methods declared in the class Class.

classAttributes(Class, Attributes)
   Extracts a list, Attributes, of the attributes of the class Class. Each attribute is represented as a functor application of the form attribute(AttributeName, AttributeContents), where AttributeName is the name of the attribute. The format of the attributes contents is unspecified.

classDefiningLoader(Class, Loader)
   Extracts the defining class loader, Loader, of the class Class.

isBootstrapLoader(Loader)
   True iff the class loader Loader is the bootstrap class loader.

methodName(Method, Name)
   Extracts the name, Name, of the method Method.

methodAccessFlags(Method, AccessFlags)
   Extracts the access flags, AccessFlags, of the method Method.

methodDescriptor(Method, Descriptor)
   Extracts the descriptor, Descriptor, of the method Method.

methodAttributes(Method, Attributes)
   Extracts a list, Attributes, of the attributes of the method Method.
isNotFinal(\text{Method}, \text{Class})

True iff \text{Method} in class \text{Class} is not final.

isProtected(\text{MemberClass}, \text{MemberName}, \text{MemberDescriptor})

True iff there is a member named \text{MemberName} with descriptor \text{MemberDescriptor} in the class \text{MemberClass} and it is protected.

isNotProtected(\text{MemberClass}, \text{MemberName}, \text{MemberDescriptor})

True iff there is a member named \text{MemberName} with descriptor \text{MemberDescriptor} in the class \text{MemberClass} and it is not protected.

samePackageName(\text{Class1}, \text{Class2})

True iff the package names of \text{Class1} and \text{Class2} are the same.

differentPackageName(\text{Class1}, \text{Class2})

True iff the package names of \text{Class1} and \text{Class2} are different.

\textbf{Specified Accessors and Utilities:} We define accessor and utility rules that extract necessary information from the representation of the class and its methods.

An environment is a six-tuple consisting of:

- a class
- a method
- the declared return type of the method
- the instructions in a method
- the maximal size of the operand stack
- a list of exception handlers

\begin{verbatim}
maxOperandStackLength(Environment, MaxStack) :-
    Environment = environment(_Class, _Method, _ReturnType,
                               _Instructions, MaxStack, _Handlers).

exceptionHandlers(Environment, Handlers) :-
    Environment = environment(_Class, _Method, _ReturnType,
                               _Instructions, _, Handlers).
\end{verbatim}
thisMethodReturnType(Environment, ReturnType) :-
    Environment = environment(_Class, _Method, ReturnType,
    _Instructions, _, _).

thisClass(Environment, class(ClassName, L)) :-
    Environment = environment(Class, _Method, _ReturnType,
    _Instructions, _,_),
    classDefiningLoader(Class, L),
    classClassName(Class, ClassName).

allInstructions(Environment, Instructions) :-
    Environment = environment(_Class, _Method, _ReturnType,
    Instructions, _, _).

offsetStackFrame(Environment, Offset, StackFrame) :-
    allInstructions(Environment, Instructions),
    member(stackMap(Offset, StackFrame), Instructions).

currentClassLoader(Environment, Loader) :-
    thisClass(Environment, class(_, Loader)).

notMember(_, []).
notMember(X, [A | More]) :- X \= A, notMember(X, More).

sameRuntimePackage(Class1, Class2) :-
    classDefiningLoader(Class1, L),
    classDefiningLoader(Class2, L),
    samePackageName(Class1, Class2).

differentRuntimePackage(Class1, Class2) :-
    classDefiningLoader(Class1, L1),
    classDefiningLoader(Class2, L2),
    L1 \= L2.

differentRuntimePackage(Class1, Class2) :-
    differentPackageName(Class1, Class2).
Abstract methods and native methods are considered to be type safe if they do not override a final method.

\[
\text{methodIsTypeSafe(Class, Method)} \leftarrow \\
\text{doesNotOverrideFinalMethod(Class, Method),} \\
\text{methodAccessFlags(Method, AccessFlags),} \\
\text{member(abstract, AccessFlags).}
\]

\[
\text{methodIsTypeSafe(Class, Method)} \leftarrow \\
\text{doesNotOverrideFinalMethod(Class, Method),} \\
\text{methodAccessFlags(Method, AccessFlags),} \\
\text{member(native, AccessFlags).}
\]

\[
\text{doesNotOverrideFinalMethod(class(‘java/lang/Object’, L), Method)} \leftarrow \\
\text{isBootstrapLoader(L).}
\]

\[
\text{doesNotOverrideFinalMethod(Class, Method)} \leftarrow \\
\text{classSuperClassName(Class, SuperclassName),} \\
\text{classDefiningLoader(Class, L),} \\
\text{loadedClass(SuperclassName, L, Superclass),} \\
\text{classMethods(Superclass, MethodList),} \\
\text{finalMethodNotOverridden(Method, Superclass, MethodList).}
\]

\[
\text{finalMethodNotOverridden(Method, Superclass, MethodList)} \leftarrow \\
\text{methodName(Method, Name),} \\
\text{methodDescriptor(Method, Descriptor),} \\
\text{member(method(\_, Name, Descriptor), MethodList),} \\
\text{isNotFinal(Method, Superclass).}
\]
finalMethodNotOverridden(Method, Superclass, MethodList) :-
    methodName(Method, Name),
    methodDescriptor(Method, Descriptor),
    notMember(method(_, Name, Descriptor), MethodList),
    doesNotOverrideFinalMethod(Superclass, Method).

4.10.1.3.3 Checking Code

Non-abstract, non-native methods are type correct if they have code and
the code is type correct.

methodIsTypeSafe(Class, Method) :-
    doesNotOverrideFinalMethod(Class, Method),
    methodAccessFlags(Method, AccessFlags),
    methodAttributes(Method, Attributes),
    notMember(native, AccessFlags),
    notMember(abstract, AccessFlags),
    member(attribute('Code', _), Attributes),
    methodWithCodeIsTypeSafe(Class, Method).

A method with code is type safe if it is possible to merge the code and
the stack frames into a single stream such that each stack map precedes
the instruction it corresponds to, and the merged stream is type correct.

methodWithCodeIsTypeSafe(Class, Method) :-
    parseCodeAttribute(Class, Method, FrameSize, MaxStack,
        ParsedCode, Handlers, StackMap),
    mergeStackMapAndCode(StackMap, ParsedCode, MergedCode),
    methodInitialStackFrame(Class, Method, FrameSize, StackFrame, ReturnType),
    Environment = environment(Class, Method, ReturnType, MergedCode, MaxStack, Handlers),
    handlersAreLegal(Environment),
    mergedCodeIsTypeSafe(Environment, MergedCode, StackFrame).
The initial type state of a method consists of an empty operand stack and local variable types derived from the type of `this` and the arguments, as well as the appropriate flag, depending on whether this is an `<init>` method.

```
methodInitialStackFrame(Class, Method, FrameSize, 
    frame(Locals, [], Flags), ReturnType):-
    methodDescriptor(Method, Descriptor),
    parseMethodDescriptor(Descriptor, RawArgs, ReturnType),
    expandTypeList(RawArgs, Args),
    methodInitialThisType(Class, Method, ThisList),
    flags(ThisList, Flags),
    append(ThisList, Args, ThisArgs),
    expandToLength(ThisArgs, FrameSize, top, Locals).

flags([uninitializedThis], [flagThisUninit]).
flags(X, []) :- X \= [uninitializedThis].

expandToLength(List, Size, _Filler, List) :- length(List, Size).
expandToLength(List, Size, Filler, Result) :-
    length(List, ListLength),
    ListLength < Size,
    Delta is Size - ListLength,
    length(Extra, Delta),
    checklist(=(Filler), Extra),
    append(List, Extra, Result).
```
For a static method this is irrelevant; the list is empty. For an instance method, we get the type of this and put it in a list.

```
methodInitialThisType(_Class, Method, []) :-
    methodAccessFlags(Method, AccessFlags),
    member(static, AccessFlags),
    methodName(Method, MethodName),
    MethodName \= '<init>'.
```

```
methodInitialThisType(Class, Method, [This]) :-
    methodAccessFlags(Method, AccessFlags),
    notMember(static, AccessFlags),
    instanceMethodInitialThisType(Class, Method, This).
```

In the <init> method of Object, the type of this is Object. In other <init> methods, the type of this is uninitializedThis. Otherwise, the type of this in an instance method is class(N, L), where N is the name of the class containing the method and L is its defining class loader.

```
instanceMethodInitialThisType(Class, Method, class('java/lang/Object', L)) :-
    methodName(Method, '<init>'),
    classDefiningLoader(Class, L),
    isBootstrapLoader(L),
    classClassName(Class, 'java/lang/Object').
```

```
instanceMethodInitialThisType(Class, Method, uninitializedThis) :-
    methodName(Method, '<init>'),
    classClassName(Class, ClassName),
    classDefiningLoader(Class, CurrentLoader),
    superclassChain(ClassName, CurrentLoader, Chain),
    Chain \= [].
```
instanceMethodInitialThisType(Class, Method, class(ClassName, L)) :-
    methodName(Method, MethodName),
    MethodName \= ’<init>’,
    classDefiningLoader(Class, L),
    classClassName(Class, ClassName).

Below are the rules for iterating through the code stream. The
assumption is that the stream is a well formed mixture of instructions
and stack maps, such that the stack map for bytecode index N appears
just before instruction N. The rules for building this mixed stream are
given later, by the predicate mergeStackMapAndCode.

The special marker aftergoto is used to indicate an unconditional
branch. If we have an unconditional branch at the end of the code, stop.

mergedCodeIsTypeSafe(_, [endOfCode(Offset)], afterGoto).

After an unconditional branch, if we have a stack map giving the type
state for the following instructions, we can proceed and typecheck them
using the type state provided by the stack map.

mergedCodeIsTypeSafe(Environment,
    [stackMap(Offset, MapFrame) | MoreCode],
    afterGoto):-
    mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).

If we have a stack map and an incoming type state, the type state must
be assignable to the one in the stack map. We may then proceed to type
check the rest of the stream with the type state given in the stack map.

mergedCodeIsTypeSafe(Environment,
    [stackMap(Offset, MapFrame) | MoreCode],
    frame(Locals, OperandStack, Flags)) :-
    frameIsAssignable(frame(Locals, OperandStack, Flags),
                     MapFrame),
    mergedCodeIsTypeSafe(Environment, MoreCode, MapFrame).
It is illegal to have code after an unconditional branch without a stack map frame being provided for it.

\[
\text{mergedCodeIsTypeSafe}(_\text{Environment},
\quad \text{[instruction(\_\_\_\_\_,\_\_) | \_\_\_\_\_\_\_ \_MoreCode]},
\quad \text{afterGoto}) :-
\]
\[
\text{write}\_\text{ln}('\text{No stack frame after unconditional branch'},
\quad \text{fail}.
\]

A merged code stream is type safe relative to an incoming type state \(T\) if it begins with an instruction \(I\) that is type safe relative to \(T\), and \(I\) satisfies its exception handlers, and the tail of the stream is type safe given the type state following that execution of \(I\).

\(\text{NextStackFrame}\) indicates what falls through to the following instruction. \(\text{ExceptionStackFrame}\) indicates what is passed to exception handlers.

\[
\text{mergedCodeIsTypeSafe}(\text{Environment},
\quad \text{[instruction(Offset,Parse) | MoreCode]},
\quad \text{frame(Locals, OperandStack, Flags)}) :-
\]
\[
\text{instructionIsTypeSafe}(\text{Parse}, \text{Environment}, \text{Offset},
\quad \text{frame(Locals, OperandStack, Flags),}
\quad \text{NextStackFrame, ExceptionStackFrame}),
\quad \text{instructionSatisfiesHandlers}(\text{Environment}, \text{Offset},
\quad \text{ExceptionStackFrame}),
\quad \text{mergedCodeIsTypeSafe}(\text{Environment, MoreCode,}
\quad \text{NextStackFrame}).
\]

Branching to a target is type safe if the target has an associated stack frame, \(\text{Frame}\), and the current stack frame, \(\text{StackFrame}\), is assignable to \(\text{Frame}\).

\[
\text{targetIsTypeSafe}(\text{Environment, StackFrame, Target}) :-
\quad \text{offsetStackFrame}(\text{Environment, Target, Frame}),
\quad \text{frameIsAssignable}(\text{StackFrame, Frame}).
\]
4.10.1.3.4 Combining Streams of Stack Maps and Instructions

Merging an empty StackMap and a list of instructions yields the original list of instructions.

\[\text{mergeStackMapAndCode([], CodeList, CodeList)}.\]

Given a list of stack map frames beginning with the type state for the instruction at \(\text{Offset}\), and a list of instructions beginning at \(\text{Offset}\), the merged list is the head of the stack frame list, followed by the head of the instruction list, followed by the merge of the tails of the two lists.

\[\text{mergeStackMapAndCode}([\text{stackMap(Offset, Map)} \mid \text{RestMap}],
\text{[instruction(Offset, Parse)} \mid \text{RestCode}],
\text{[stackMap(Offset, Map), instruction(Offset, Parse)} \mid \text{RestMerge}]) \leftarrow
\text{mergeStackMapAndCode(RestMap, RestCode, RestMerge)}.\]

Otherwise, given a list of stack frames beginning with the type state for the instruction at \(\text{OffsetM}\), and a list of instructions beginning at \(\text{OffsetP}\), then, if \(\text{OffsetP} < \text{OffsetM}\), the merged list consists of the head of the instruction list, followed by the merge of the stack frame list and the tail of the instruction list.

\[\text{mergeStackMapAndCode}([\text{stackMap(OffsetM, Map)} \mid \text{RestMap}],
\text{[instruction(OffsetP, Parse)} \mid \text{RestCode}],
\text{[instruction(OffsetP, Parse)} \mid \text{RestMerge}]) \leftarrow
\text{OffsetP} < \text{OffsetM},
\text{mergeStackMapAndCode([stackMap(OffsetM, Map)} \mid \text{RestMap}],
\text{RestCode, RestMerge}).\]

Otherwise, the merge of the two lists is undefined. Since the instruction list has monotonically increasing offsets, the merge of the two lists is not defined unless every stack map frame offset has a corresponding instruction offset and the stack map frames are in monotonically increasing order.
4.10.1.3.5 Exception Handling

An instruction *satisfies its exception handlers* if it satisfies every exception handler that is applicable to the instruction.

```
instructionSatisfiesHandlers(Environment, Offset, ExceptionStackFrame) :-
    exceptionHandlers(Environment, Handlers),
    sublist(isApplicableHandler(Offset), Handlers, ApplicableHandlers),
    checklist(instructionSatisfiesHandler(Environment, ExceptionStackFrame), ApplicableHandlers).
```

An exception handler is *applicable* to an instruction if the offset of the instruction is greater or equal to the start of the handler’s range and less than the end of the handler’s range.

```
isApplicableHandler(Offset, handler(Start, End, _Target, _ClassName)) :-
    Offset >= Start,
    Offset < End.
```

An instruction *satisfies* an exception handler if its incoming type state is `StackFrame`, and the handler’s target (the initial instruction of the handler code) is type safe assuming an incoming type state T. The type state T is derived from `StackFrame` by replacing the operand stack with a stack whose sole element is the handler’s exception class.

```
instructionSatisfiesHandler(Environment, StackFrame, Handler) :-
    Handler = handler(_, _, Target, _),
    currentClassLoader(Environment, CurrentLoader),
    handlerExceptionClass(Handler, ExceptionClass, CurrentLoader),
    /* The stack consists of just the exception. */
    StackFrame = frame(Locals, _, Flags),
    ExcStackFrame = frame(Locals, [ ExceptionClass ], Flags),
    operandStackHasLegalLength(Environment, ExcStackFrame),
    targetIsTypeSafe(Environment, ExcStackFrame, Target).
```
The exception class of a handler is **Throwable** if the handlers class entry is 0, otherwise it is the class named in the handler.

```
handlerExceptionClass(handler(_, _, _, 0),
    class('java/lang/Throwable', BL), _) :-
    isBootstrapLoader(BL).
handlerExceptionClass(handler(_, _, _, Name), class(Name, L), L) :-
    Name \= 0.
```

An exception handler is legal if its start (Start) is less than its end (End), there exists an instruction whose offset is equal to Start, there exists an instruction whose offset equals End and the handler’s exception class is assignable to the class **Throwable**.

```
handlersAreLegal(Environment) :-
    exceptionHandlers(Environment, Handlers),
    checklist(handlerIsLegal(Environment), Handlers).
handlerIsLegal(Environment, Handler) :-
    Handler = handler(Start, End, Target, _),
    Start < End,
    allInstructions(Environment, Instructions),
    member(instruction(Start, _), Instructions),
    offsetStackFrame(Environment, Target, _),
    instructionsIncludeEnd(Instructions, End),
    currentClassLoader(Environment, CurrentLoader),
    handlerExceptionClass(Handler, ExceptionClass, CurrentLoader),
    isBootstrapLoader(BL),
    isAssignable(ExceptionClass, class('java/lang/Throwable', BL)).
```

```
instructionsIncludeEnd(Instructions, End) :-
    member(instruction(End, _), Instructions).
instructionsIncludeEnd(Instructions, End) :-
    member(endOfCode(End), Instructions).
```
4.10.1.4 Instructions

4.10.1.4.1 Isomorphic Instructions

Many bytecodes have type rules that are completely isomorphic to the rules for other bytecodes. If a bytecode $b_1$ is isomorphic to another bytecode $b_2$, then the type rule for $b_1$ is the same as the type rule for $b_2$.

```
instructionIsTypeSafe(Instruction, Environment, Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    instructionHasEquivalentTypeRule(Instruction, IsomorphicInstruction),
    instructionIsTypeSafe(IsomorphicInstruction, Environment, Offset, StackFrame, NextStackFrame, ExceptionStackFrame).
```

4.10.1.4.2 Manipulating the Operand Stack

This section defines the rules for legally manipulating the type state’s operand stack. Manipulation of the operand stack is complicated by the fact that some types occupy two entries on the stack. The predicates given in this section take this into account, allowing the rest of the specification to abstract from this issue.

```
\text{canPop(frame}(Locals, \text{OperandStack}, Flags), Types,\
    \text{frame}(Locals, \text{PoppedOperandStack}, Flags)) :-\
    popMatchingList(OperandStack, Types, PoppedOperandStack).
\text{popMatchingList(OperandStack}, [], \text{OperandStack}).
\text{popMatchingList(OperandStack}, [P | \text{Rest}], \text{NewOperandStack}) :-\n    popMatchingType(OperandStack, P, TempOperandStack, _ActualType),
    popMatchingList(TempOperandStack, Rest, NewOperandStack).
```
Pop an individual type off the stack. More precisely, if the logical top of the stack is some subtype of the specified type, \texttt{Type}, then pop it. If a type occupies two stack slots, the logical top of stack type is really the type just below the top, and the top of stack is the unusable type \texttt{top}.

\begin{verbatim}
popMatchingType([ActualType | OperandStack], Type, OperandStack, ActualType) :-
    sizeOf(Type, 1),
    isAssignable(ActualType, Type).
popMatchingType([top, ActualType | OperandStack], Type, OperandStack, ActualType) :-
    sizeOf(Type, 2),
    isAssignable(ActualType, Type).
\end{verbatim}

Push a logical type onto the stack. The exact behavior varies with the size of the type. If the pushed type is of size 1, we just push it onto the stack. If the pushed type is of size 2, we push it, and then push \texttt{top}.

\begin{verbatim}
pushOperandStack(OperandStack, 'void', OperandStack).
pushOperandStack(OperandStack, Type, [Type | OperandStack]) :-
    sizeOf(Type, 1).
pushOperandStack(OperandStack, Type, [top, Type | OperandStack]) :-
    sizeOf(Type, 2).
\end{verbatim}

The length of the operand stack must not exceed the declared maximum stack length.

\begin{verbatim}
operandStackHasLegalLength(Environment, OperandStack) :-
    length(OperandStack, Length),
    maxOperandStackLength(Environment, MaxStack),
    Length =< MaxStack.
\end{verbatim}
Category 1 types occupy a single stack slot. Popping a logical type of category 1, Type, off the stack is possible if the top of the stack is Type and Type is not top (otherwise it could denote the upper half of a category 2 type). The result is the incoming stack, with the top slot popped off.

\[
popCategory1([\text{Type} | \text{Rest}], \text{Type}, \text{Rest}) :-
\]
\[
\text{Type} \neq \text{top},
\text{sizeOf(}\text{Type}, 1).\]

Category 2 types occupy two stack slots. Popping a logical type of category 2, Type, off the stack is possible if the top of the stack is type top, and the slot directly below it is Type. The result is the incoming stack, with the top 2 slots popped off.

\[
popCategory2([\text{top}, \text{Type} | \text{Rest}], \text{Type}, \text{Rest}) :-
\]
\[
\text{sizeOf(}\text{Type}, 2).\]

canSafelyPush(\text{Environment}, \text{InputOperandStack}, \text{Type},
\text{OutputOperandStack}) :-
pushOperandStack(\text{InputOperandStack}, \text{Type}, \text{OutputOperandStack}),
operandStackHasLegalLength(\text{Environment}, \text{OutputOperandStack}).

canSafelyPushList(\text{Environment}, \text{InputOperandStack}, \text{Types},
\text{OutputOperandStack}) :-
\]
\[
canPushList(\text{InputOperandStack}, \text{Types}, \text{OutputOperandStack}),
operandStackHasLegalLength(\text{Environment}, \text{OutputOperandStack}).
\]
canPushList(\text{InputOperandStack}, [\text{Type} | \text{Rest}], \text{OutputOperandStack}) :-
pushOperandStack(\text{InputOperandStack}, \text{Type}, \text{InterimOperandStack}),
canPushList(\text{InterimOperandStack}, \text{Rest}, \text{OutputOperandStack}).
canPushList(\text{InputOperandStack}, [], \text{InputOperandStack}).
4.10.1.4.3 Loads

All load instructions are variations on a common pattern, varying the type of the value that the instruction loads.

Loading a value of type \texttt{Type} from local variable \texttt{Index} is type safe, if the type of that local variable is \texttt{ActualType}, \texttt{ActualType} is assignable to \texttt{Type}, and pushing \texttt{ActualType} onto the incoming operand stack is a valid type transition that yields a new type state \texttt{NextStackFrame}. After execution of the load instruction, the type state will be \texttt{NextStackFrame}.

\begin{verbatim}
loadIsTypeSafe(\_Environment, Index, Type, StackFrame, NextStackFrame) :-
  StackFrame = frame(Locals, _OperandStack, _Flags),
  nth0(Index, Locals, ActualType),
  isAssignable(ActualType, Type),
  validTypeTransition(\_Environment, [], ActualType, StackFrame,
                      NextStackFrame).
\end{verbatim}

4.10.1.4.4 Stores

All store instructions are variations on a common pattern, varying the type of the value that the instruction stores.

In general, a store instruction is type safe if the local variable it references is of a type that is a supertype of \texttt{Type}, and the top of the operand stack is of a subtype of \texttt{Type}, where \texttt{Type} is the type the instruction is designed to store.

More precisely, the store is type safe if one can pop a type \texttt{ActualType} that “matches” \texttt{Type} (i.e., is a subtype of \texttt{Type}) off the operand stack, and then legally assign that type the local variable \texttt{LIndex}.

\begin{verbatim}
storeIsTypeSafe(\_Environment, Index, Type,
               frame(Locals, OperandStack, Flags),
               frame(NextLocals, NextOperandStack, Flags)) :-
  popMatchingType(OperandStack, Type,
                  NextOperandStack, ActualType),
  modifyLocalVariable(Index, ActualType, Locals, NextLocals).
\end{verbatim}
Given local variables Locals, modifying Index to have type Type results in the local variable list NewLocals. The modifications are somewhat involved, because some values (and their corresponding types) occupy two local variables. Hence, modifying LN may require modifying LN+1 (because the type will occupy both the N and N+1 slots) or LN-1 (because local N used to be the upper half of the two word value/type starting at local N-1, and so local N-1 must be invalidated), or both. This is described further below. We start at L0 and count up.

modifyLocalVariable(Index, Type, Locals, NewLocals) :-
    modifyLocalVariable(0, Index, Type, Locals, NewLocals).

Given the suffix of the local variable list starting at index I, LocalsRest, modifying local variable Index to have type Type results in the local variable list suffix NewLocalsRest.

If I < Index - 1, just copy the input to the output and recurse forward. If I = Index - 1, the type of local I may change. This can occur if L1 has a type of size 2. Once we set L1+1 to the new type (and the corresponding value), the type/value of L1 will be invalidated, as its upper half will be trashed. Then we recurse forward.

When we find the variable, and it only occupies one word, we change it to Type and we’re done. When we find the variable, and it occupies two words, we change its type to Type and the next word to top.

modifyLocalVariable(I, Index, Type, [Locals1 | LocalsRest],
    [Locals1 | NextLocalsRest] ) :-
    I < Index - 1,
    I1 is I + 1,
    modifyLocalVariable(I1, Index, Type, LocalsRest, NextLocalsRest).

modifyLocalVariable(I, Index, Type, [Locals1 | LocalsRest],
    [NextLocals1 | NextLocalsRest] ) :-
    I =:= Index - 1,
    modifyPreIndexVariable(Locals1, NextLocals1),
    modifyLocalVariable(Index, Index, Type, LocalsRest, NextLocalsRest).
modifyLocalVariable(Index, Index, Type, [_, _ | LocalsRest],
                      [Type | LocalsRest]) :-
                      sizeof(Type, 1).

modifyLocalVariable(Index, Index, Type, [_, _ | LocalsRest],
                      [Type, top | LocalsRest]) :-
                      sizeof(Type, 2).

We refer to a local whose index immediately precedes a local whose
type will be modified as a pre-index variable. The future type of a pre-
index variable of type InputType is Result. If the type, Value, of the pre-
index local is of size 1, it doesn’t change. If the type of the pre-index
local, Value, is 2, we need to mark the lower half of its two word value
as unusable, by setting its type to top.

modifyPreIndexVariable(Type, Type) :- sizeof(Type, 1).
modifyPreIndexVariable(Type, top) :- sizeof(Type, 2).

Given a list of types, this clause produces a list where every type of size
2 has been substituted by two entries: one for itself, and one top entry.
The result then corresponds to the representation of the list as 32 bit
words in the Java virtual machine.

expandTypeList([], []).
expandTypeList([Item | List], [Item | Result]) :-
                      sizeof(Item, 1),
                      expandTypeList(List, Result).
expandTypeList([Item | List], [Item, top | Result]) :-
                      sizeof(Item, 2),
                      expandTypeList(List, Result).
### 4.10.1.4.5 List of all Instructions

In general, the type rule for an instruction is given relative to an environment **Environment** that defines the class and method in which the instruction occurs, and the offset **Offset** within the method at which the instruction occurs. The rule states that if the incoming type state **StackFrame** fulfills certain requirements, then:

- The instruction is type safe.
- It is provable that the type state after the instruction completes normally has a particular form given by **NextStackFrame**, and that the type state after the instruction completes abruptly is given by **ExceptionStackFrame**.

The natural language description of the rule is intended to be readable, intuitive and concise. As such, the description avoids repeating all the contextual assumptions given above. In particular:

- We do not explicitly mention the environment.
- When we speak of the operand stack or local variables in the following, we are referring to the operand stack and local variable components of a type state: either the incoming type state or the outgoing one.
- The type state after the instruction completes abruptly is almost always identical to the incoming type state. We only discuss the type state after the instruction completes abruptly when that is not the case.
- We speak of popping and pushing types onto the operand stack. We do not explicitly discuss issues of stack underflow or overflow, but assume that these operations can be completed successfully. The formal rules for operand stack manipulation ensure that the necessary checks are made.
- Similarly, the text discusses only the manipulation of logical types. In practice, some types take more than one word. We abstract from these representation details in our discussion, but the logical rules that manipulate data do not.

Any ambiguities can be resolved by referring to the formal Prolog rules.
**aaload:**

An *aaload* instruction is type safe iff one can validly replace types matching `int` and an array type with component type `ComponentType` where `ComponentType` is a subtype of `Object`, with `ComponentType` yielding the outgoing type state.

```prolog
instructionIsTypeSafe(aaload, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    nth1OperandStackIs(2, StackFrame, ArrayType),
    arrayComponentType(ArrayType, ComponentType),
    isBootstrapLoader(BL),
    validTypeTransition(Environment, [int, arrayOf(class('java/lang/Object', BL))], ComponentType, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The component type of an array of `X` is `X`. We define the component type of `null` to be `null`.

```prolog
arrayComponentType(arrayOf(X), X).
arrayComponentType(null, null).
```

**aastore:**

An *aastore* instruction is type safe iff one can validly pop types matching `Object`, `int`, and an array of `Object` off the incoming operand stack yielding the outgoing type state.

```prolog
instructionIsTypeSafe(aastore, _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    isBootstrapLoader(BL),
    canPop(StackFrame, [class('java/lang/Object', BL), int, arrayOf(class('java/lang/Object', BL))], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
aconst_null:

An `aconst_null` instruction is type safe if one can validly push the type `null` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(aconst_null, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], null, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

aload:

An `aload` instruction with operand `Index` is type safe and yields an outgoing type state `NextStackFrame`, if a load instruction with operand `Index` and type `reference` is type safe and yields an outgoing type state `NextStackFrame`.

```
instructionIsTypeSafe(aload(Index), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, reference, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

aload_<n>:

The instructions `aload_<n>`, for `0 ≤ n ≤ 3`, are typesafe iff the equivalent `aload` instruction is type safe.

```
instructionHasEquivalentTypeRule(aload_0,aload(0)).
instructionHasEquivalentTypeRule(aload_1,aload(1)).
instructionHasEquivalentTypeRule(aload_2,aload(2)).
instructionHasEquivalentTypeRule(aload_3,aload(3)).
```
**anewarray:**

An `anewarray` instruction with operand $CP$ is type safe iff $CP$ refers to a constant pool entry denoting either a class type or an array type, and one can legally replace a type matching `int` on the incoming operand stack with an array with component type $CP$ yielding the outgoing type state.

```
instructionIsTypeSafe(anewarray(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    (CP = class(_, _) ; CP = arrayOf(_)),
    validTypeTransition(Environment, [int], arrayOf(CP), StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**areturn:**

An `areturn` instruction is type safe iff the enclosing method has a declared return type, $ReturnType$, that is a reference type, and one can validly pop a type matching $ReturnType$ off the incoming operand stack.

```
instructionIsTypeSafe(areturn, Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, ReturnType),
    isAssignable(ReturnType, reference),
    canPop(StackFrame, [ReturnType], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
arraylength:

An `arraylength` instruction is type safe iff one can validly replace an array type on the incoming operand stack with the type `int` yielding the outgoing type state.

```
instructionIsTypeSafe(arraylength, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  nth1OperandStackIs(1, StackFrame, ArrayType),
  arrayComponentType(ArrayType, _),
  validTypeTransition(Environment, [top], int, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

astore:

An `astore` instruction with operand `Index` is type safe and yields an outgoing type state `NextStackFrame`, if a store instruction with operand `Index` and type `reference` is type safe and yields an outgoing type state `NextStackFrame`.

```
instructionIsTypeSafe(astore(Index), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  storeIsTypeSafe(Environment, Index, reference, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

astore_<n>:

The instructions `astore_<n>`, for $0 \leq n \leq 3$, are typesafe iff the equivalent `astore` instruction is type safe.

```
instructionHasEquivalentTypeRule(astore_0,astore(0)).
instructionHasEquivalentTypeRule(astore_1,astore(1)).
instructionHasEquivalentTypeRule(astore_2,astore(2)).
instructionHasEquivalentTypeRule(astore_3,astore(3)).
```
athrow:

An athrow instruction is type safe iff the top of the operand stack matches Throwable.

instructionIsTypeSafe(athrow, _Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
  isBootstrapLoader(BL),
  canPop(StackFrame, [class('java/lang/Throwable', BL)], _PoppedStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).

baload:

A baload instruction is type safe iff one can validly replace types matching int and a small array type on the incoming operand stack with int yielding the outgoing type state.

instructionIsTypeSafe(baload, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  nth1OperandStackIs(2, StackFrame, Array),
  isSmallArray(Array),
  validTypeTransition(Environment, [int, top], int, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).

An array type is a small array type if it is an array of byte, an array of boolean, or a subtype thereof (null).

isSmallArray(arrayOf(byte)).
isSmallArray(arrayOf(boolean)).
isSmallArray(null).
**bastore:**

A **bastore** instruction is type safe iff one can validly pop types matching `int`, `int` and a small array type off the incoming operand stack yielding the outgoing type state.

\[
\text{instructionIsTypeSafe(bastore, \_Environment, \_Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-}
\]
\[
\text{nth1OperandStackIs(3, StackFrame, Array),}
\text{isSmallArray(Array),}
\text{canPop(StackFrame, [int, int, top], NextStackFrame),}
\text{exceptionStackFrame(StackFrame, ExceptionStackFrame).}
\]

**bipush:**

A **bipush** instruction is type safe iff the equivalent **sipush** instruction is type safe

\[
\text{instructionHasEquivalentTypeRule(bipush(Value), sipush(Value)).}
\]

**caload:**

A **caload** instruction is type safe iff one can validly replace types matching `int` and array of `char` on the incoming operand stack with `int` yielding the outgoing type state.

\[
\text{instructionIsTypeSafe(caload, Environment, \_Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-}
\]
\[
\text{validTypeTransition(Environment, [int, array\{char\}], int, StackFrame, NextStackFrame),}
\text{exceptionStackFrame(StackFrame, ExceptionStackFrame).}
\]
**castore:**

A *castore* instruction is type safe iff one can validly pop types matching `int`, `int` and array of `char` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(castore, _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  canPop(StackFrame, [int, int, arrayOf(char)], NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**checkcast:**

A *checkcast* instruction with operand `CP` is type safe iff `CP` refers to a constant pool entry denoting either a class or an array, and one can validly replace the type `Object` on top of the incoming operand stack with the type denoted by `CP` yielding the outgoing type state.

```
instructionIsTypeSafe(checkcast(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  (CP = class(_, _) ; CP = arrayOf(_)),
  isBootstrapLoader(BL),
  validTypeTransition(Environment, [class('java/lang/Object', BL)], CP, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
d2f:

A d2f instruction is type safe if one can validly pop `double` off the incoming operand stack and replace it with `float`, yielding the outgoing type state.

```
instructionIsTypeSafe(d2f, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2i:

A d2i instruction is type safe if one can validly pop `double` off the incoming operand stack and replace it with `int`, yielding the outgoing type state.

```
instructionIsTypeSafe(d2i, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

d2l:

A d2l instruction is type safe if one can validly pop `double` off the incoming operand stack and replace it with `long`, yielding the outgoing type state.

```
instructionIsTypeSafe(d2l, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
**dadd:**

A *dadd* instruction is type safe iff one can validly replace types matching `double` and `double` on the incoming operand stack with `double` yielding the outgoing type state.

```prolog
instructionIsTypeSafe(dadd, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double, double], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**daload:**

A *daload* instruction is type safe iff one can validly replace types matching `int` and array of `double` on the incoming operand stack with `double` yielding the outgoing type state.

```prolog
instructionIsTypeSafe(daload, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, arrayOf(double)], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**dastore:**

A *dastore* instruction is type safe iff one can validly pop types matching `double`, `int` and array of `double` off the incoming operand stack yielding the outgoing type state.

```prolog
instructionIsTypeSafe(dastore, _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [double, int, arrayOf(double)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```


dcmp<op>:

A `dcmpg` instruction is type safe iff one can validly replace types matching `double` and `double` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(dcmpg, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [double, double], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A `dcmpl` instruction is type safe iff the equivalent `dcmpg` instruction is type safe.

```
instructionHasEquivalentTypeRule(dcmpl, dcmpg).
```

dconst_<d>:

A `dconst_0` instruction is type safe if one can validly push the type `double` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(dconst_0, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A `dconst_1` instruction is type safe iff the equivalent `dconst_0` instruction is type safe.

```
instructionHasEquivalentTypeRule(dconst_1, dconst_0).
```
**ddiv:**

A *ddiv* instruction is type safe iff the equivalent *dadd* instruction is type safe.

**instructionHasEquivalentTypeRule(ddiv, dadd).**

**dload:**

A *dload* instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a load instruction with operand Index and type **double** is type safe and yields an outgoing type state NextStackFrame.

**instructionIsTypeSafe(dload(Index), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
loadIsTypeSafe(Environment, Index, double, StackFrame, NextStackFrame),
exceptionStackFrame(StackFrame, ExceptionStackFrame).**

**dload_<n>:**

The instructions *dload_<n>, for 0 ≤ n ≤ 3*, are typesafe iff the equivalent *dload* instruction is type safe.

**instructionHasEquivalentTypeRule(dload_0,dload(0)).**
**instructionHasEquivalentTypeRule(dload_1,dload(1)).**
**instructionHasEquivalentTypeRule(dload_2,dload(2)).**
**instructionHasEquivalentTypeRule(dload_3,dload(3)).**

**dmul:**

A *dmul* instruction is type safe iff the equivalent *dadd* instruction is type safe.

**instructionHasEquivalentTypeRule(dmul, dadd).**
dneg:

A dneg instruction is type safe iff there is a type matching double on the incoming operand stack. The dneg instruction does not alter the type state.

```
instructionIsTypeSafe(dneg, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  validTypeTransition(Environment, [double], double, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

drem:

A drem instruction is type safe iff the equivalent dadd instruction is type safe.

```
instructionHasEquivalentTypeRule(drem, dadd).
```

dreturn:

A dreturn instruction is type safe if the enclosing method has a declared return type of double, and one can validly pop a type matching double off the incoming operand stack.

```
instructionIsTypeSafe(dreturn, Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
  thisMethodReturnType(Environment, double),
  canPop(StackFrame, [double], _PoppedStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
**dstore:**

A `dstore` instruction with operand `Index` is type safe and yields an outgoing type state `NextStackFrame`, if a store instruction with operand `Index` and type `double` is type safe and yields an outgoing type state `NextStackFrame`.

```
instructionIsTypeSafe(dstore(Index), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**dstore_<n>:**

The instructions `dstore_<n>`, for `0 ≤ n ≤ 3`, are type safe iff the equivalent `dstore` instruction is type safe.

```
instructionHasEquivalentTypeRule(dstore_0, dstore(0)).
instructionHasEquivalentTypeRule(dstore_1, dstore(1)).
instructionHasEquivalentTypeRule(dstore_2, dstore(2)).
instructionHasEquivalentTypeRule(dstore_3, dstore(3)).
```

**dsub:**

A `dsub` instruction is type safe iff the equivalent `dadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(dsub, dadd).
```
**dup:**

A **dup** instruction is type safe iff one can validly replace a category 1 type, `Type`, with the types `Type`, `Type`, yielding the outgoing type state.

```prolog
instructionIsTypeSafe(dup, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    popCategory1(InputOperandStack, Type, _),
    canSafelyPush(Environment, InputOperandStack, Type, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**dup_x1:**

A **dup_x1** instruction is type safe iff one can validly replace two category 1 types, `Type1` and `Type2`, on the incoming operand stack with the types `Type1`, `Type2`, `Type1`, yielding the outgoing type state.

```prolog
instructionIsTypeSafe(dup_x1, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1], OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
dup_x2:

A dup_x2 instruction is type safe iff it is a type safe form of the dup_x2 instruction.

instructionIsTypeSafe(dup_x2, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Local, InputOperandStack, Flags),
    dup_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Local, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).

A dup_x2 instruction is a type safe form of the dup_x2 instruction iff it is a type safe form 1 dup_x2 instruction or a type safe form 2 dup_x2 instruction.

dup_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).

dup_x2SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    dup_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack).
A `dup_x2` instruction is a type safe form 1 `dup_x2` instruction iff one can validly replace three category 1 types, `Type1`, `Type2`, `Type3` on the incoming operand stack with the types `Type1`, `Type2`, `Type3`, `Type1`, yielding the outgoing type state.

```
dup_x2Form1IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory1(Stack1, Type2, Stack2),
    popCategory1(Stack2, Type3, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type3, Type2, Type1], OutputOperandStack).
```

A `dup_x2` instruction is a type safe form 2 `dup_x2` instruction iff one can validly replace a category 1 type, `Type1`, and a category 2 type, `Type2`, on the incoming operand stack with the types `Type1`, `Type2`, `Type1`, yielding the outgoing type state.

```
dup_x2Form2IsTypeSafe(Environment, InputOperandStack, OutputOperandStack) :-
    popCategory1(InputOperandStack, Type1, Stack1),
    popCategory2(Stack1, Type2, Rest),
    canSafelyPushList(Environment, Rest, [Type1, Type2, Type1], OutputOperandStack).
```
**dup2:**

A `dup2` instruction is type safe iff it is a type safe form of the `dup2` instruction.

\[
\text{instructionIsTypeSafe}(\text{dup2}, \text{Environment}, \_\text{Offset}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) :\]
\[
\text{StackFrame} = \text{frame}(\text{Locals}, \text{InputOperandStack}, \text{Flags}),
\text{dup2SomeFormIsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}),
\text{NextStackFrame} = \text{frame}(\text{Locals}, \text{OutputOperandStack}, \text{Flags}),
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]

A `dup2` instruction is a type safe form of the `dup2` instruction iff it is a type safe form 1 `dup2` instruction or a type safe form 2 `dup2` instruction.

\[
\text{dup2SomeFormIsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}) :\]
\[
\text{dup2Form1IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}).
\text{dup2SomeFormIsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}) :\]
\[
\text{dup2Form2IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}).
\]
A **dup2** instruction is a *type safe form 1 dup2 instruction* iff one can validly replace two category 1 types, \( \text{Type1} \) and \( \text{Type2} \) on the incoming operand stack with the types \( \text{Type1}, \text{Type2}, \text{Type1}, \text{Type2} \), yielding the outgoing type state.

\[
dup2\text{Form1IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \\
\text{OutputOperandStack}):- \\
pop\text{Category1}(\text{InputOperandStack}, \text{Type1}, \text{TempStack}), \\
pop\text{Category1}(\text{TempStack}, \text{Type2}, \_), \\
can\text{SafelyPushList}(\text{Environment}, \text{InputOperandStack}, \\
\quad[\text{Type1}, \text{Type2}], \text{OutputOperandStack}).
\]

A **dup2** instruction is a *type safe form 2 dup2 instruction* iff one can validly replace a category 2 type, \( \text{Type} \) on the incoming operand stack with the types \( \text{Type}, \text{Type} \), yielding the outgoing type state.

\[
dup2\text{Form2IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \\
\text{OutputOperandStack}):- \\
pop\text{Category2}(\text{InputOperandStack}, \text{Type}, \_), \\
can\text{SafelyPush}(\text{Environment}, \text{InputOperandStack}, \\
\quad\text{Type}, \text{OutputOperandStack}).
\]
**dup2_x1:**

A `dup2_x1` instruction is type safe iff it is a type safe form of the `dup2_x1` instruction.

```
instructionIsTypeSafe(dup2_x1, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, InputOperandStack, Flags),
    dup2_x1SomeFormIsTypeSafe(Environment, InputOperandStack, OutputOperandStack),
    NextStackFrame = frame(Locals, OutputOperandStack, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A `dup2_x1` instruction is a type safe form of the `dup2_x1` instruction iff it is a type safe form 1 `dup2_x1` instruction or a type safe form 2 `dup2_x2` instruction.

```
dup2_x1SomeFormIsTypeSafe( Environment, InputOperandStack, OutputOperandStack ) :-
    dup2_x1Form1IsTypeSafe( Environment, InputOperandStack, OutputOperandStack ).

dup2_x1SomeFormIsTypeSafe( Environment, InputOperandStack, OutputOperandStack ) :-
    dup2_x1Form2IsTypeSafe( Environment, InputOperandStack, OutputOperandStack ).
```
A \texttt{dup2\_x1} instruction is \textit{a type safe form 1} \texttt{dup2\_x1} instruction iff one can validly replace three category 1 types, \texttt{Type1}, \texttt{Type2}, \texttt{Type3}, on the incoming operand stack with the types \texttt{Type1}, \texttt{Type2}, \texttt{Type3}, \texttt{Type1}, \texttt{Type2}, yielding the outgoing type state.

\begin{verbatim}
dup2\_x1Form1IsTypeSafe(\texttt{Environment}, \texttt{InputOperandStack},
                          \texttt{OutputOperandStack}) :-
  popCategory1(\texttt{InputOperandStack}, \texttt{Type1}, \texttt{Stack1}),
  popCategory1(\texttt{Stack1}, \texttt{Type2}, \texttt{Stack2}),
  popCategory1(\texttt{Stack2}, \texttt{Type3}, \texttt{Rest}),
  canSafelyPushList(\texttt{Environment}, \texttt{Rest},
                     [\texttt{Type2}, \texttt{Type1}, \texttt{Type3}, \texttt{Type2}, \texttt{Type1}],
                     \texttt{OutputOperandStack}).
\end{verbatim}

A \texttt{dup2\_x1} instruction is \textit{a type safe form 2} \texttt{dup2\_x1} instruction iff one can validly replace a category 2 type, \texttt{Type1}, and a category 1 type, \texttt{Type2}, on the incoming operand stack with the types \texttt{Type1}, \texttt{Type2}, \texttt{Type1}, yielding the outgoing type state.

\begin{verbatim}
dup2\_x1Form2IsTypeSafe(\texttt{Environment}, \texttt{InputOperandStack},
                          \texttt{OutputOperandStack}) :-
  popCategory2(\texttt{InputOperandStack}, \texttt{Type1}, \texttt{Stack1}),
  popCategory1(\texttt{Stack1}, \texttt{Type2}, \texttt{Rest}),
  canSafelyPushList(\texttt{Environment}, \texttt{Rest},
                     [\texttt{Type1}, \texttt{Type2}, \texttt{Type1}],
                     \texttt{OutputOperandStack}).
\end{verbatim}
**dup2_x2:**

A `dup2_x2` instruction is type safe iff it is a type safe form of the `dup2_x2` instruction.

\[
\text{instructionIsTypeSafe}(\text{dup2}_x2, \text{Environment}, _\text{Offset}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) ::=
\]

\[
\text{StackFrame} = \text{frame}(\text{Locals}, \text{InputOperandStack}, \text{Flags}),
\]

\[
\text{dup2}_x2\text{SomeFormIsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}),
\]

\[
\text{NextStackFrame} = \text{frame}(\text{Locals}, \text{OutputOperandStack}, \text{Flags}),
\]

\[
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]

A `dup2_x2` instruction is a type safe form of the `dup2_x2` instruction iff one of the following holds:

- it is a type safe form 1 `dup2_x2` instruction.
- it is a type safe form 2 `dup2_x2` instruction.
- it is a type safe form 3 `dup2_x2` instruction.
- it is a type safe form 4 `dup2_x2` instruction.

\[
\text{dup2}_x2\text{SomeFormIsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}) ::=
\]

\[
\text{dup2}_x2\text{Form1IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}).
\]

\[
\text{dup2}_x2\text{Form1IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}) ::=
\]

\[
\text{dup2}_x2\text{Form2IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}).
\]

\[
\text{dup2}_x2\text{Form2IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}) ::=
\]

\[
\text{dup2}_x2\text{Form3IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}).
\]

\[
\text{dup2}_x2\text{Form3IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}) ::=
\]

\[
\text{dup2}_x2\text{Form4IsTypeSafe}(\text{Environment}, \text{InputOperandStack}, \text{OutputOperandStack}).
\]
A `dup2_x2` instruction is a type safe form 1 `dup2_x2` instruction iff one can validly replace four category 1 types, Type1, Type2, Type3, Type4, on the incoming operand stack with the types Type1, Type2, Type3, Type4, Type1, Type2, yielding the outgoing type state.

A `dup2_x2` instruction is a type safe form 2 `dup2_x2` instruction iff one can validly replace a category 2 type, Type1, and two category 1 types, Type2, Type3, on the incoming operand stack with the types Type1, Type2, Type3, Type1, yielding the outgoing type state.
A **dup2_x2** instruction is a *type safe form 3 dup2_x2 instruction* iff one can validly replace two category 1 types, **Type1**, **Type2**, and a category 2 type, **Type3**, on the incoming operand stack with the types **Type1**, **Type2**, **Type3**, **Type1**, **Type2**, yielding the outgoing type state.

\[
\text{dup2_x2Form3IsTypeSafe}(\text{Environment, InputOperandStack, OutputOperandStack}) :- \\
\text{popCategory1}(\text{InputOperandStack, Type1, Stack1}), \\
\text{popCategory1}(\text{Stack1, Type2, Stack2}), \\
\text{popCategory2}(\text{Stack2, Type3, Rest}), \\
\text{canSafelyPushList}(\text{Environment, Rest, [Type2, Type1, Type3, Type2, Type1], OutputOperandStack}).
\]

A **dup2_x2** instruction is a *type safe form 4 dup2_x2 instruction* iff one can validly replace two category 2 types, **Type1**, **Type2**, on the incoming operand stack with the types **Type1**, **Type2**, **Type1**, yielding the outgoing type state.

\[
\text{dup2_x2Form4IsTypeSafe}(\text{Environment, InputOperandStack, OutputOperandStack}) :- \\
\text{popCategory2}(\text{InputOperandStack, Type1, Stack1}), \\
\text{popCategory2}(\text{Stack1, Type2, Rest}), \\
\text{canSafelyPushList}(\text{Environment, Rest, [Type1, Type2, Type1], OutputOperandStack}).
\]
f2d:

An f2d instruction is type safe if one can validly pop float off the incoming operand stack and replace it with double, yielding the outgoing type state.

\[
\text{instructionIsTypeSafe}(f2d, \text{Environment}, \_\text{Offset}, \text{StackFrame}, \\
\quad \text{NextStackFrame, ExceptionStackFrame}) :- \\
\quad \text{validTypeTransition}(\text{Environment}, [\text{float}], \text{double}, \\
\quad \quad \text{StackFrame, NextStackFrame}), \\
\quad \text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]

f2i:

An f2i instruction is type safe if one can validly pop float off the incoming operand stack and replace it with int, yielding the outgoing type state.

\[
\text{instructionIsTypeSafe}(f2i, \text{Environment}, \_\text{Offset}, \text{StackFrame}, \\
\quad \text{NextStackFrame, ExceptionStackFrame}) :- \\
\quad \text{validTypeTransition}(\text{Environment}, [\text{float}], \text{int}, \\
\quad \quad \text{StackFrame, NextStackFrame}), \\
\quad \text{exceptionStackFrame}(\text{StackFrame, ExceptionStackFrame}).
\]

f2l:

An f2l instruction is type safe if one can validly pop float off the incoming operand stack and replace it with long, yielding the outgoing type state.

\[
\text{instructionIsTypeSafe}(f2l, \text{Environment}, \_\text{Offset}, \text{StackFrame}, \\
\quad \text{NextStackFrame, ExceptionStackFrame}) :- \\
\quad \text{validTypeTransition}(\text{Environment}, [\text{float}], \text{long}, \\
\quad \quad \text{StackFrame, NextStackFrame}), \\
\quad \text{exceptionStackFrame}(\text{StackFrame, ExceptionStackFrame}).
\]
fadd:

An fadd instruction is type safe iff one can validly replace types matching float and float on the incoming operand stack with float yielding the outgoing type state.

\[
\text{instructionIsTypeSafe}(\text{fadd}, \text{Environment}, \_\text{Offset}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) ::-
\]
\[
\text{validTypeTransition}(\text{Environment}, [\text{float}, \text{float}], \text{float}, \text{StackFrame}, \text{NextStackFrame}),
\]
\[
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]

faload:

An faload instruction is type safe iff one can validly replace types matching int and array of float on the incoming operand stack with float yielding the outgoing type state.

\[
\text{instructionIsTypeSafe}(\text{faload}, \text{Environment}, \_\text{Offset}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) ::-
\]
\[
\text{validTypeTransition}(\text{Environment}, [\text{int}, \text{arrayOf(float)}], \text{float}, \text{StackFrame}, \text{NextStackFrame}),
\]
\[
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]

fastore:

An fastore instruction is type safe iff one can validly pop types matching float, int and array of float off the incoming operand stack yielding the outgoing type state.

\[
\text{instructionIsTypeSafe}(\text{fastore}, \_\text{Environment}, \_\text{Offset}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) ::-
\]
\[
\text{canPop}(\text{StackFrame}, [\text{float}, \text{int}, \text{arrayOf(float)}], \text{NextStackFrame}),
\]
\[
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]
fcmp<op>:

An `fcmpg` instruction is type safe iff one can validly replace types matching `float` and `float` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(fcmpg, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float, float], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

An `fcmpl` instruction is type safe iff the equivalent `fcmpg` instruction is type safe.

```
instructionHasEquivalentTypeRule(fcmpl, fcmpg).
```

fconst_<f>:

An `fconst_0` instruction is type safe if one can validly push the type `float` onto the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(fconst_0, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

The rules for the other variants of `fconst` are equivalent:

```
instructionHasEquivalentTypeRule(fconst_1, fconst_0).
instructionHasEquivalentTypeRule(fconst_2, fconst_0).
```
**fdiv:**

An `fdiv` instruction is type safe iff the equivalent `fadd` instruction is type safe.

`instructionHasEquivalentTypeRule(fdiv, fadd).`

**fload:**

An `fload` instruction with operand `Index` is type safe and yields an outgoing type state `NextStackFrame`, if a load instruction with operand `Index` and type `float` is type safe and yields an outgoing type state `NextStackFrame`.

```
instructionIsTypeSafe(fload(Index), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :- 
  loadIsTypeSafe(Environment, Index, float, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**fload_<n>:**

The instructions `fload_<n>`, for $0 \leq n \leq 3$, are typesafe iff the equivalent `fload` instruction is type safe.

```
instructionHasEquivalentTypeRule(fload_0,fload(0)).
instructionHasEquivalentTypeRule(fload_1,fload(1)).
instructionHasEquivalentTypeRule(fload_2,fload(2)).
instructionHasEquivalentTypeRule(fload_3,fload(3)).
```

**fmul:**

An `fmul` instruction is type safe iff the equivalent `fadd` instruction is type safe.

`instructionHasEquivalentTypeRule(fmul, fadd).`
fneg:

An \texttt{fneg} instruction is type safe iff there is a type matching \texttt{float} on the incoming operand stack. The \texttt{fneg} instruction does not alter the type state.

\begin{verbatim}
instructionIsTypeSafe(fneg, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [float], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}

frem:

An \texttt{frem} instruction is type safe iff the equivalent \texttt{fadd} instruction is type safe.

\begin{verbatim}
instructionHasEquivalentTypeRule(frem, fadd).
\end{verbatim}

freturn:

An \texttt{freturn} instruction is type safe if the enclosing method has a declared return type of \texttt{float}, and one can validly pop a type matching \texttt{float} off the incoming operand stack.

\begin{verbatim}
instructionIsTypeSafe(freturn, Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, float),
    canPop(StackFrame, [float], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}
fstore:

An fstore instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type float is type safe and yields an outgoing type state NextStackFrame.

instructionIsTypeSafe(fstore(Index), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).

fstore_<n>:

The instructions fstore_<n>, for 0 ≤ n ≤ 3, are typesafe iff the equivalent fstore instruction is type safe.

instructionHasEquivalentTypeRule(fstore_0,fstore(0)).
instructionHasEquivalentTypeRule(fstore_1,fstore(1)).
instructionHasEquivalentTypeRule(fstore_2,fstore(2)).
instructionHasEquivalentTypeRule(fstore_3,fstore(3)).

fsub:

An fsub instruction is type safe iff the equivalent fadd instruction is type safe.

instructionHasEquivalentTypeRule(fsub, fadd).
getfield:

A getfield instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is FieldType, declared in a class FieldClass, and one can validly replace a type matching FieldClass with type FieldType on the incoming operand stack yielding the outgoing type state. FieldClass must not be an array type. Protected fields are subject to additional checks.

\[
\text{instructionIsTypeSafe(getfield(CP), Environment, \_Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-}
\]
\[
\begin{align*}
\text{CP} & = \text{field(FieldClass, FieldName, FieldDescriptor)}, \\
\text{parseFieldDescriptor(FieldDescriptor, FieldType)}, \\
\text{passesProtectedCheck(Environment, FieldClass, FieldName, FieldDescriptor, StackFrame)}, \\
\text{validTypeTransition(Environment, [class(FieldClass)], FieldType, StackFrame, NextStackFrame)}, \\
\text{exceptionStackFrame(StackFrame, ExceptionStackFrame)}. \\
\end{align*}
\]

The protected check applies only to members of superclasses of the current class. Other cases will be caught by the access checking done at resolution time. If the name of a class is not the name of any superclass, it cannot be a superclass, and so it can safely be ignored.

\[
\text{passesProtectedCheck(Environment, MemberClassName, MemberName, MemberDescriptor, StackFrame) :-}
\]
\[
\begin{align*}
\text{thisClass(Environment, class(CurrentClassName, CurrentLoader))}, \\
\text{superclassChain(CurrentClassName, CurrentLoader, Chain)}, \\
\text{notMember(class(MemberClassName, \_), Chain)}. \\
\end{align*}
\]
Using a superclass member that is not protected is trivially correct.

If the `MemberClassName` is the same as the name of a superclass, the class being resolved may indeed be a superclass. In this case, if no superclass named `MemberClassName` in a different runtime package has a protected member named `MemberName` with descriptor `MemberDescriptor`, the protected check need not apply.

This is because the actual class being resolved will either be one of these superclasses, in which case we know that it is either in the same runtime package, and the access is legal; or the member in question is not protected and the check does not apply; or it will be a subclass, in which case the check would succeed anyway; or it will be some other class in the same runtime package, in which case the access is legal and the check need not take place; or the verifier need not flag this as a problem, since it will be caught anyway because resolution will per force fail.

```prolog
passesProtectedCheck(Env, MemberClassName, MemberName, MemberDescriptor, Frame) :-
    thisClass(Env, class(ClassName, _)),
    superclassChain(ClassName, _, Chain),
    member(class(MemberClassName, _), Chain),
    classesInOtherPkgWithProtectedMember(
        class(ClassName, _), MemberName, MemberDescriptor, MemberClassName, Chain, []).```
If there does exist a protected superclass member in a different runtime package, then load MemberClassName; if the member in question is not protected, the check does not apply.

```prolog
passesProtectedCheck(Environment, MemberClassName, MemberName, MemberDescriptor, frame(_Locals, [Target | Rest], _Flags)) :-
    thisClass(Environment, class(CurrentClassName, CurrentLoader)),
    superclassChain(CurrentClassName, CurrentLoader, Chain),
    member(class(MemberClassName, _), Chain),
    classesInOtherPkgWIthProtectedMember(
        class(CurrentClassName, CurrentLoader), MemberName,
        MemberDescriptor, MemberClassName, Chain, List),
    List /= [],
    loadedClass(MemberClassName, CurrentLoader, ReferencedClass),
    isNotProtected(ReferencedClass, MemberName, MemberDescriptor).
```
Otherwise, use of a member of an object of type Target requires that Target be assignable to the type of the current class.

passesProtectedCheck(Environment, MemberClassName, MemberName, MemberDescriptor, frame(_Locals, [Target | Rest], _Flags)) :-
    thisClass(Environment, class(CurrentClassName, CurrentLoader)),
    superclassChain(CurrentClassName, CurrentLoader, Chain),
    member(class(MemberClassName, _), Chain),
    classesInOtherPkgWIthProtectedMember(class(CurrentClassName, CurrentLoader),
                                          MemberName, MemberDescriptor, MemberClassName, Chain, List),
    List /= [],
    loadedClass(MemberClassName, CurrentLoader, ReferencedClass),
    isProtected(ReferencedClass, MemberName, MemberDescriptor),
    isAssignable(Target, class(CurrentClassName, CurrentLoader)).

superclassChain(ClassName, L, [class(SuperclassName, Ls) | Rest]) :-
    loadedClass(ClassName, L, Class),
    classSuperclassName(Class, SuperclassName),
    classDefiningLoader(Class, Ls),
    superclassChain(SuperclassName, Ls, Rest).

superclassChain('java/lang/Object', L, []) :-
    loadedClass('java/lang/Object', L, Class),
    classDefiningLoader(Class, BL),
    isBootstrapLoader(BL).
The predicate `classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, Chain, List)` is true if `List` is the set of classes in `Chain` with name `MemberClassName` that are in a different runtime package than `Class` which have a protected member named `MemberName` with descriptor `MemberDescriptor`.

```prolog
classesInOtherPkgWithProtectedMember(_, _, _, _, [], []).
classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, [class(MemberClassName, L) | Tail], [class(MemberClassName, L) | T]) :-
    differentRuntimePackage(Class, class(MemberClassName, L)),
    loadedClass(MemberClassName, L, Super),
    isProtected(Super, MemberName, MemberDescriptor),
    classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, Tail, T).

classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, [class(MemberClassName, L) | Tail], T) :-
    differentRuntimePackage(Class, class(MemberClassName, L)),
    loadedClass(MemberClassName, L, Super),
    isNotProtected(Super, MemberName, MemberDescriptor).
    classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, Tail, T).

classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, [class(MemberClassName, L) | Tail], T) :-
    sameRuntimePackage(Class, class(MemberClassName, L)),
    classesInOtherPkgWithProtectedMember(Class, MemberName, MemberDescriptor, MemberClassName, Tail, T).
```
getstatic:

A getstatic instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is FieldType, and one can validly push FieldType on the incoming operand stack yielding the outgoing type state.

\[
\text{instructionIsTypeSafe(getstatic(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame)} \ :
\]

\[
\begin{align*}
\text{CP} &= \text{field}(_\text{FieldClass}, _\text{FieldName}, \text{FieldDescriptor}), \\
\text{parseFieldDescriptor(}\text{FieldDescriptor, FieldType),} \\
\text{validTypeTransition(}\text{Environment, [], FieldType, StackFrame, NextStackFrame),} \\
\text{exceptionStackFrame(}\text{StackFrame, ExceptionStackFrame}).
\end{align*}
\]

goto:

A goto instruction is type safe iff its target operand is a valid branch target.

\[
\text{instructionIsTypeSafe(goto(Target), Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame)} \ :
\]

\[
\begin{align*}
\text{targetIsTypeSafe(}\text{Environment, StackFrame, Target),} \\
\text{exceptionStackFrame(}\text{StackFrame, ExceptionStackFrame}).
\end{align*}
\]

goto_w:

A goto_w instruction is type safe iff the equivalent goto instruction is type safe.

\[
\text{instructionHasEquivalentTypeRule(goto_w(Target), goto(Target)).}
\]
i2b:

An i2b instruction is type safe iff the equivalent ineg instruction is type safe.

instructionHasEquivalentTypeRule(i2b, ineg).

i2c:

An i2c instruction is type safe iff the equivalent ineg instruction is type safe.

instructionHasEquivalentTypeRule(i2c, ineg).

i2d:

An i2d instruction is type safe if one can validly pop int off the incoming operand stack and replace it with double, yielding the outgoing type state.

instructionIsTypeSafe(i2d, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int], double, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).

i2f:

An i2f instruction is type safe if one can validly pop int off the incoming operand stack and replace it with float, yielding the outgoing type state.

instructionIsTypeSafe(i2f, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int], float, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
i2l:

An i2l instruction is type safe if one can validly pop int off the incoming operand stack and replace it with long, yielding the outgoing type state.

\[
\text{instructionIsTypeSafe(i2l, Environment, \_Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-}
\text{validTypeTransition(Environment, [int], long, StackFrame, NextStackFrame),}
\text{exceptionStackFrame(StackFrame, ExceptionStackFrame).}
\]

i2s:

An i2s instruction is type safe iff the equivalent ineg instruction is type safe.

\[
\text{instructionHasEquivalentTypeRule(i2s, ineg).}
\]

iadd:

An iadd instruction is type safe iff one can validly replace types matching int and int on the incoming operand stack with int yielding the outgoing type state.

\[
\text{instructionIsTypeSafe(iadd, Environment, \_Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-}
\text{validTypeTransition(Environment, [int, int], int, StackFrame, NextStackFrame),}
\text{exceptionStackFrame(StackFrame, ExceptionStackFrame).}
\]
**iaload:**

An `iaload` instruction is type safe iff one can validly replace types matching `int` and array of `int` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(iaload, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  validTypeTransition(Environment, [int, arrayOf(int)], int, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**iand:**

An `iand` instruction is type safe iff the equivalent `iadd` instruction is type safe.

```
instructionHasEquivalentTypeRule(iand, iadd).
```

**iastore:**

An `iastore` instruction is type safe iff one can validly pop types matching `int`, `int` and array of `int` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(iastore, _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  canPop(StackFrame, [int, int, arrayOf(int)], NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
if_acmp<cond>:

An if_acmpeq instruction is type safe iff one can validly pop types matching reference and reference on the incoming operand stack yielding the outgoing type state NextStackFrame, and the operand of the instruction, Target, is a valid branch target assuming an incoming type state of NextStackFrame.

instructionIsTypeSafe(if_acmpeq(Target), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  canPop(StackFrame, [reference, reference], NextStackFrame),
  targetIsTypeSafe(Environment, NextStackFrame, Target),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).

The rule for if_acmp_ne is identical.

instructionHasEquivalentTypeRule(if_acmpne(Target), if_acmpeq(Target)).

if_icmp<cond>:

An if_icmpeq instruction is type safe iff one can validly pop types matching int and int on the incoming operand stack yielding the outgoing type state NextStackFrame, and the operand of the instruction, Target, is a valid branch target assuming an incoming type state of NextStackFrame.

instructionIsTypeSafe(if_icmpeq(Target), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  canPop(StackFrame, [int, int], NextStackFrame),
  targetIsTypeSafe(Environment, NextStackFrame, Target),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
The rules for all other variants of the `if_icmp` instruction are identical

\[
\text{instructionHasEquivalentTypeRule}(\text{if_icmpge}(\text{Target}), \text{if_icmpeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{if_icmpgt}(\text{Target}), \text{if_icmpeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{if_icmple}(\text{Target}), \text{if_icmpeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{if_icmplt}(\text{Target}), \text{if_icmpeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{if_icmpne}(\text{Target}), \text{if_icmpeq}(\text{Target})).
\]

**if_<cond>:**

An `if_eq` instruction is type safe iff one can validly pop a type matching `int` off the incoming operand stack yielding the outgoing type state `NextStackFrame`, and the operand of the instruction, `Target`, is a valid branch target assuming an incoming type state of `NextStackFrame`.

\[
\text{instructionIsTypeSafe}(\text{ifeq}(\text{Target}), \text{Environment}, \_\text{Offset}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) :\text{-}
\]
\[
\text{canPop}(\text{StackFrame}, [\text{int}], \text{NextStackFrame}),
\]
\[
\text{targetIsTypeSafe}(\text{Environment}, \text{NextStackFrame}, \text{Target}),
\]
\[
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]

The rules for all other variations of the `if<cond>` instruction are identical

\[
\text{instructionHasEquivalentTypeRule}(\text{ifge}(\text{Target}), \text{ifeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{ifgt}(\text{Target}), \text{ifeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{ifle}(\text{Target}), \text{ifeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{iflt}(\text{Target}), \text{ifeq}(\text{Target})).
\]
\[
\text{instructionHasEquivalentTypeRule}(\text{ifne}(\text{Target}), \text{ifeq}(\text{Target})).
\]
ifnonnull:

An ifnonnull instruction is type safe iff one can validly pop a type matching reference off the incoming operand stack yielding the outgoing type state NextStackFrame, and the operand of the instruction, Target, is a valid branch target assuming an incoming type state of NextStackFrame.

instructionIsTypeSafe(ifnonnull(Target), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  canPop(StackFrame, [reference], NextStackFrame),
  targetIsTypeSafe(Environment, NextStackFrame, Target),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).

ifnull:

An ifnull instruction is type safe iff the equivalent ifnonnull instruction is type safe.

instructionHasEquivalentTypeRule(ifnull(Target), ifnonnull(Target)).

iinc:

An iinc instruction with first operand Index is type safe iff LIndex has type int. The iinc instruction does not change the type state.

instructionIsTypeSafe(iinc(Index, _Value), _Environment, _Offset, StackFrame, StackFrame, ExceptionStackFrame) :-
  StackFrame = frame(Locals, _OperandStack, _Flags),
  nth0(Index, Locals, int),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
**iload:**

An \texttt{iload} instruction with operand \texttt{Index} is type safe and yields an outgoing type state \texttt{NextStackFrame}, if a load instruction with operand \texttt{Index} and type \texttt{int} is type safe and yields an outgoing type state \texttt{NextStackFrame}.

\begin{verbatim}
instructionIsTypeSafe(iload(Index), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}

**iload\_<n>:**

The instructions \texttt{iload\_<n>}, for \(0 \leq n \leq 3\), are typesafe iff the equivalent \texttt{iload} instruction is type safe.

\begin{verbatim}
instructionHasEquivalentTypeRule(iload_0,iload(0)).
instructionHasEquivalentTypeRule(iload_1,iload(1)).
instructionHasEquivalentTypeRule(iload_2,iload(2)).
instructionHasEquivalentTypeRule(iload_3,iload(3)).
\end{verbatim}

**imul:**

An \texttt{imul} instruction is type safe iff the equivalent \texttt{iadd} instruction is type safe.

\begin{verbatim}
instructionHasEquivalentTypeRule(imul, iadd).
\end{verbatim}
ineg:

An ineg instruction is type safe iff there is a type matching int on the incoming operand stack. The ineg instruction does not alter the type state.

\[
\text{instructionIsTypeSafe}(\text{ineg}, \text{Environment}, _{\text{Offset}}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) :- \\
\text{validTypeTransition}(\text{Environment}, [\text{int}], \text{int}, \text{StackFrame}, \text{NextStackFrame}), \\
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]

instanceof:

An instanceof instruction with operand CP is type safe iff CP refers to a constant pool entry denoting either a class or an array, and one can validly replace the type Object on top of the incoming operand stack with type int yielding the outgoing type state.

\[
\text{instructionIsTypeSafe}(\text{instanceof}(\text{CP}), \text{Environment}, _{\text{Offset}}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) :- \\
(\text{CP} = \text{class}(\_, \_); \text{CP} = \text{arrayOf}(\_)), \\
\text{isBootstrapLoader}(\text{BL}), \\
\text{validTypeTransition}(\text{Environment}, [\text{class}(\text{'java/lang/Object'}), \text{BL}], \text{int}, \text{StackFrame}, \text{NextStackFrame}), \\
\text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}).
\]
An `invokeinterface` instruction is type safe iff all of the following conditions hold:

- Its first operand, `CP`, refers to a constant pool entry denoting an interface method named `MethodName` with descriptor `Descriptor` that is a member of an interface `MethodClassName`.
- `MethodName` is not `<init>`.
- `MethodName` is not `<clinit>`.
- Its second operand, `Count`, is a valid count operand (see below).
- One can validly replace types matching the type `MethodClassName` and the argument types given in `Descriptor` on the incoming operand stack with the return type given in `Descriptor`, yielding the outgoing type state.

```prolog
instructionIsTypeSafe(invokeinterface(CP, Count, 0), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  CP = imethod(MethodClassName, MethodName, Descriptor),
  MethodName \= '<init>',
  MethodName \= '<clinit>',
  parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
  currentClassLoader(Environment, L),
  reverse([class(MethodClassName, L) | OperandArgList], StackArgList),
  canPop(StackFrame, StackArgList, TempFrame),
  validTypeTransition(Environment, [], ReturnTyype, TempFrame, NextStackFrame),
  countIsValid(Count, StackFrame, TempFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
The count operand of an `invokeinterface` instruction is valid if it equals the size of the arguments to the instruction. This is equal to the difference between the size of `InputFrame` and `OutputFrame`.

```prolog
countIsValid(Count, InputFrame, OutputFrame) :-
    InputFrame = frame(_Locals1, OperandStack1, _Flags1),
    OutputFrame = frame(_Locals2, OperandStack2, _Flags2),
    length(OperandStack1, Length1),
    length(OperandStack2, Length2),
    Count =:= Length1 - Length2.
```

**invokespecial:**

An `invokespecial` instruction is type safe iff all of the following conditions hold:

- Its first operand, `CP`, refers to a constant pool entry denoting a method named `MethodName` with descriptor `Descriptor` that is a member of a class `MethodClassName`.

Either

- `MethodName` is not `<init>`.  
- `MethodName` is not `<clinit>`.  

- One can validly replace types matching the current class and the argument types given in `Descriptor` on the incoming operand stack with the return type given in `Descriptor`, yielding the outgoing type state.

- One can validly replace types matching the class `MethodClassName` and the argument types given in `Descriptor` on the incoming operand stack with the return type given in `Descriptor`.  

instructionIsTypeSafe(invokespecial(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-

CP = method(MethodClass, MethodName, Descriptor),
MethodName \= ’<init>’,
MethodName \= ’<clinit>’,
parsedMethodDescriptor(Descriptor, OperandArgList, ReturnType),
thisClass(Environment, CurrentClass),
reverse([CurrentClass | OperandArgList], StackArgList),
validTypeTransition(Environment, StackArgList, ReturnType, StackFrame, NextStackFrame),
currentClassLoader(Environment, L),
reverse([class(MethodClass, L) | OperandArgList], StackArgList2),
validTypeTransition(Environment, StackArgList2, ReturnType, StackFrame, _ResultStackFrame),
isAssignableFrom(class(CurrentClass, L),
class(MethodClass, L)).

exceptionStackFrame(StackFrame, ExceptionStackFrame).
Or

- **MethodName** is `<init>`.
- **Descriptor** specifies a **void** return type.
- One can validly pop types matching the argument types given in **Descriptor** and an uninitialized type, **UninitializedArg**, off the incoming operand stack, yielding **OperandStack**.
- The outgoing type state is derived from the incoming type state by first replacing the incoming operand stack with **OperandStack** and then replacing all instances of **UninitializedArg** with the type of instance being initialized.

```
instructionIsTypeSafe(invokespecial(CP), Environment, _Offset,
                      StackFrame, NextStackFrame,
                      ExceptionStackFrame) :-
  CP = method(MethodClassName, '<init>', Descriptor),
  parseMethodDescriptor(Descriptor, OperandArgList, void),
  reverse(OperandArgList, StackArgList),
  canPop(StackFrame, StackArgList, TempFrame),
  TempFrame = frame(Locals, FullOperandStack, Flags),
  FullOperandStack = [UninitializedArg | OperandStack],
  currentClassLoader(Environment, CurrentLoader),
  rewrittenUninitializedType(UninitializedArg, Environment,
                            class(MethodClassName, CurrentLoader), This),
  rewrittenInitializationFlags(UninitializedArg, Flags, NextFlags),
  substitute(UninitializedArg, This, OperandStack,
            NextOperandStack),
  substitute(UninitializedArg, This, Locals,
            NextLocals),
  NextStackFrame = frame(NextLocals, NextOperandStack,
                         NextFlags),
  ExceptionStackFrame = frame(NextLocals, [], Flags),
  passesProtectedCheck(Environment, MethodClassName, '<init>',
                        Descriptor, NextStackFrame).
```
Special rule for `invokespecial` of an `<init>` method.

This rule is the sole motivation for passing back a distinct exception stack frame. The concern is that `invokespecial` can cause a superclass `<init>` method to be invoked, and that invocation could fail, leaving `this` uninitialized. This situation cannot be created using source code in the Java programming language, but can be created by programming in bytecode directly.

The original frame holds an uninitialized object in a local and has flag `uninitializedThis`. Normal termination of `invokespecial` initializes the uninitialized object and turns off the `uninitializedThis` flag. But if the invocation of an `<init>` method throws an exception, the uninitialized object might be left in a partially initialized state, and needs to be made permanently unusable. This is represented by an exception frame containing the broken object (the new value of the local) and the `uninitializedThis` flag (the old flag). There is no way to get from an apparently-initialized object bearing the `uninitializedThis` flag to a properly initialized object, so the object is permanently unusable. If not for this case, the exception stack frame could be the same as the input stack frame.

```
rewrittenUninitializedType(uninitializedThis, Environment, _MethodClass, This) :-
    thisClass(Environment, This).

rewrittenUninitializedType(uninitialized(Address), Environment, MethodClass, MethodClass) :-
    allInstructions(Environment, Instructions),
    member(instruction(Address, new(MethodClass)), Instructions).
```
Computes what type the uninitialized argument's type needs to be rewritten to.

There are 2 cases.

If we are initializing an object within its constructor, its type is initially \texttt{uninitializedThis}. This type will be rewritten to the type of the class of the \texttt{<init>} method.

The second case arises from initialization of an object created by \texttt{new}. The uninitialized arg type is rewritten to \texttt{MethodClass}, the type of the method holder of \texttt{<init>}. We check whether there really is a \texttt{new} instruction at \texttt{Address}.

\begin{verbatim}
rewrittenInitializationFlags(uninitializedThis, _Flags, []). rewrittenInitializationFlags(uninitialized(_), Flags, Flags).

substitute(_Old, _New, [], []). substitute(Old, New, [Old | FromRest], [New | ToRest]) :- substitute(Old, New, FromRest, ToRest).

substitute(Old, New, [From1 | FromRest], [From1 | ToRest]) :- From1 \neq Old, substitute(Old, New, FromRest, ToRest).
\end{verbatim}
invokestatic:

An `invokestatic` instruction is type safe iff all of the following conditions hold:

- Its first operand, `CP`, refers to a constant pool entry denoting a method named `MethodName` with descriptor `Descriptor`.
- `MethodName` is not `<clinit>`.
- One can validly replace types matching the argument types given in `Descriptor` on the incoming operand stack with the return type given in `Descriptor`, yielding the outgoing type state.

```
instructionIsTypeSafe(invokestatic(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  CP = method(_MethodClassName, MethodName, Descriptor),
  MethodName \= '<clinit>',
  parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
  reverse(OperandArgList, StackArgList),
  validTypeTransition(Environment, StackArgList, ReturnTy
  validTypeTransition(Environment, StackArgList, ReturnTy
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
invokevirtual:

An `invokevirtual` instruction is type safe iff all of the following conditions hold:

- Its first operand, `CP`, refers to a constant pool entry denoting a method named `MethodName` with descriptor `Descriptor` that is a member of a class `MethodClassName`.
- `MethodName` is not `<init>`. 
- `MethodName` is not `<clinit>`.
- One can validly replace types matching the class `MethodClassName` and the argument types given in `Descriptor` on the incoming operand stack with the return type given in `Descriptor`, yielding the outgoing type state.
- If the method is protected, the usage conforms to the special rules governing access to protected members.

```
instructionIsTypeSafe(invokevirtual(CP), Environment, _Offset, 
                      StackFrame, NextStackFrame, 
                      ExceptionStackFrame) :-
  CP = method(MethodClassName, MethodName, Descriptor),
  MethodName //= '<init>',
  MethodName //= '<clinit>',
  parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),
  reverse(OperandArgList, ArgList),
  currentClassLoader(Environment, L),
  reverse([class(MethodClassName, L) | OperandArgList], 
          StackArgList),
  validTypeTransition(Environment, StackArgList, ReturnType, 
                       StackFrame, NextStackFrame),
  canPop(StackFrame, ArgList, PoppedFrame),
  passesProtectedCheck(Environment, MethodClassName, 
                        MethodName, Descriptor, PoppedFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
ior:

An **ior** instruction is type safe iff the equivalent **iadd** instruction is type safe.

```
instructionHasEquivalentTypeRule(ior, iadd).
```

irem:

An **irem** instruction is type safe iff the equivalent **iadd** instruction is type safe.

```
instructionHasEquivalentTypeRule(irem, iadd).
```

ireturn:

An **ireturn** instruction is type safe if the enclosing method has a declared return type of **int**, and one can validly pop a type matching **int** off the incoming operand stack.

```
instructionIsTypeSafe(ireturn, Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, int),
    canPop(StackFrame, [int], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

ishl:

An **ishl** instruction is type safe iff the equivalent **iadd** instruction is type safe.

```
instructionHasEquivalentTypeRule(ishl, iadd).
```
ishr:

An ishr instruction is type safe iff the equivalent iadd instruction is type safe.

instructionHasEquivalentTypeRule(ishr, iadd).

istore:

An istore instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type int is type safe and yields an outgoing type state NextStackFrame.

instructionIsTypeSafe(istore(Index), Environment, Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    storeIsTypeSafe(Environment, Index, int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).

istore_<n>:

The instructions istore_<n>, for 0 ≤ n ≤ 3, are typesafe iff the equivalent istore instruction is type safe.

instructionHasEquivalentTypeRule(istore_0,istore(0)).
instructionHasEquivalentTypeRule(istore_1,istore(1)).
instructionHasEquivalentTypeRule(istore_2,istore(2)).
instructionHasEquivalentTypeRule(istore_3,istore(3)).

isub:

An isub instruction is type safe iff the equivalent iadd instruction is type safe.

instructionHasEquivalentTypeRule(isub, iadd).
iushr:

An **iushr** instruction is type safe iff the equivalent **iadd** instruction is type safe.

\[
\text{instructionHasEquivalentTypeRule(iushr, iadd)}.
\]

ixor:

An **ixor** instruction is type safe iff the equivalent **iadd** instruction is type safe.

\[
\text{instructionHasEquivalentTypeRule(ixor, iadd)}.
\]

l2d:

An **l2d** instruction is type safe if one can validly pop **long** off the incoming operand stack and replace it with **double**, yielding the outgoing type state.

\[
\text{instructionIsTypeSafe(l2d, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-}
\]

\[
\text{validTypeTransition(Environment, [long], double, StackFrame, NextStackFrame),}
\]

\[
\text{exceptionStackFrame(StackFrame, ExceptionStackFrame)}.
\]

l2f:

An **l2f** instruction is type safe if one can validly pop **long** off the incoming operand stack and replace it with **float**, yielding the outgoing type state.

\[
\text{instructionIsTypeSafe(l2f, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-}
\]

\[
\text{validTypeTransition(Environment, [long], float, StackFrame, NextStackFrame),}
\]

\[
\text{exceptionStackFrame(StackFrame, ExceptionStackFrame)}.
\]
l2i:

An l2i instruction is type safe if one can validly pop \texttt{long} off the incoming operand stack and replace it with \texttt{int}, yielding the outgoing type state.

\begin{verbatim}
instructionIsTypeSafe(l2i, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}

ladd:

An ladd instruction is type safe iff one can validly replace types matching \texttt{long} and \texttt{long} on the incoming operand stack with \texttt{long} yielding the outgoing type state.

\begin{verbatim}
instructionIsTypeSafe(ladd, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long, long], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}

laload:

An laaload instruction is type safe iff one can validly replace types matching \texttt{int} and array of \texttt{long} on the incoming operand stack with \texttt{long} yielding the outgoing type state.

\begin{verbatim}
instructionIsTypeSafe(laload, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, arrayOf(long)], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}
land:

An `land` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(land, ladd).
```

lastore:

A `lastore` instruction is type safe iff one can validly pop types matching `long`, `int` and array of `long` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(lastore, _Environment, _Offset, StackFrame,  
  NextStackFrame, ExceptionStackFrame) :-  
  canPop(StackFrame, [long, int, arrayOf(long)], NextStackFrame),  
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

lcmp:

A `lcmp` instruction is type safe iff one can validly replace types matching `long` and `long` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(lcmp, Environment, _Offset, StackFrame,  
  NextStackFrame, ExceptionStackFrame) :-  
  validTypeTransition(Environment, [long, long], int, StackFrame,  
    NextStackFrame),  
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
lconst_\langle l\rangle:

An lconst_0 instruction is type safe if one can validly push the type long onto the incoming operand stack yielding the outgoing type state.

instructionIsTypeSafe(lconst_0, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).

An lconst_1 instruction is type safe iff the equivalent lconst_0 instruction is type safe.

instructionHasEquivalentTypeRule(lconst_1, lconst_0).

ldc:

An ldc instruction with operand CP is type safe iff CP refers to a constant pool entry denoting an entity of type Type, where Type is either int, float, String or Class and one can validly push Type onto the incoming operand stack yielding the outgoing type state.

instructionIsTypeSafe(ldc(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    functor(CP, Tag, _),
    isBootstrapLoader(BL),
    member([Tag, Type], [
        [int, int],
        [float, float],
        [string, class('java/lang/String', BL)],
        [classConst, class('java/lang/Class', BL)]
    ]),
    validTypeTransition(Environment, [], Type, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).


**ldc\_w:**

An \texttt{ldc\_w} instruction is type safe iff the equivalent \texttt{ldc} instruction is type safe.

\begin{verbatim}
instructionHasEquivalentTypeRule(ldc\_w(CP), ldc(CP))
\end{verbatim}

**ldc2\_w:**

An \texttt{ldc2\_w} instruction with operand \texttt{CP} is type safe iff \texttt{CP} refers to a constant pool entry denoting an entity of type \texttt{Tag}, where \texttt{Tag} is either \texttt{long} or \texttt{double}, and one can validly push \texttt{Tag} onto the incoming operand stack yielding the outgoing type state.

\begin{verbatim}
instructionIsTypeSafe(ldc2\_w(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  functor(CP, Tag, _),
  member(Tag, [long, double]),
  validTypeTransition(Environment, [], Tag, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}

**ldiv:**

An \texttt{ldiv} instruction is type safe iff the equivalent \texttt{ladd} instruction is type safe.

\begin{verbatim}
instructionHasEquivalentTypeRule(ldiv, ladd).
\end{verbatim}
lload:

An \texttt{lload} instruction with operand \texttt{Index} is type safe and yields an outgoing type state \texttt{NextStackFrame}, if a load instruction with operand \texttt{Index} and type \texttt{long} is type safe and yields an outgoing type state \texttt{NextStackFrame}.

\begin{verbatim}
instructionIsTypeSafe(lload(Index), Environment, _Offset, StackFrame,
NextStackFrame, ExceptionStackFrame) :-
    loadIsTypeSafe(Environment, Index, long, StackFrame,
NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}

lload\_<n>:

The instructions \texttt{lload\_<n>}, for \(0 \leq n \leq 3\), are typesafe iff the equivalent \texttt{lload} instruction is type safe.

\begin{verbatim}
instructionHasEquivalentTypeRule(lload_0,lload(0)).
instructionHasEquivalentTypeRule(lload_1,lload(1)).
instructionHasEquivalentTypeRule(lload_2,lload(2)).
instructionHasEquivalentTypeRule(lload_3,lload(3)).
\end{verbatim}

lmul:

An \texttt{lmul} instruction is type safe iff the equivalent \texttt{ladd} instruction is type safe.

\begin{verbatim}
instructionHasEquivalentTypeRule(lmul, ladd).
\end{verbatim}
**lneg:**

An `lneg` instruction is type safe iff there is a type matching `long` on the incoming operand stack. The `lneg` instruction does not alter the type state.

```
instructionIsTypeSafe(lneg, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [long], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**lookupswitch:**

A `lookupswitch` instruction is type safe if its keys are sorted, one can validly pop `int` off the incoming operand stack yielding a new type state `BranchStackFrame`, and all of the instructions targets are valid branch targets assuming `BranchStackFrame` as their incoming type state.

```
instructionIsTypeSafe(lookupswitch(Targets, Keys), Environment, _, StackFrame, afterGoto, ExceptionStackFrame) :-
    sort(Keys, Keys),
    canPop(StackFrame, [int], BranchStackFrame),
    checklist(targetIsTypeSafe(Environment, BranchStackFrame), Targets),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**lor:**

An `lor` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(lor, ladd).
```
Irem:

An `lrem` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(lrem, ladd).
```

Ireturn:

An `lreturn` instruction is type safe if the enclosing method has a declared return type of `long`, and one can validly pop a type matching `long` off the incoming operand stack.

```
instructionIsTypeSafe(lreturn, Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
    thisMethodReturnType(Environment, long),
    canPop(StackFrame, [long], _PoppedStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

Ishl:

An `lshl` instruction is type safe if one can validly replace the types `int` and `long` on the incoming operand stack with the type `long` yielding the outgoing type state.

```
instructionIsTypeSafe(lshl, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [int, long], long, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

Ishr:

An `lshr` instruction is type safe iff the equivalent `lshl` instruction is type safe.

```
instructionHasEquivalentTypeRule(lshr, lshl).
```
lstore:

An lstore instruction with operand Index is type safe and yields an outgoing type state NextStackFrame, if a store instruction with operand Index and type long is type safe and yields an outgoing type state NextStackFrame.

\[ \text{instructionIsTypeSafe}(\text{lstore}(\text{Index}), \text{Environment}, _\text{Offset}, \text{StackFrame}, \text{NextStackFrame}, \text{ExceptionStackFrame}) \] :-
\[ \text{storeIsTypeSafe}(\text{Environment}, \text{Index}, \text{long}, \text{StackFrame}, \text{NextStackFrame}), \]
\[ \text{exceptionStackFrame}(\text{StackFrame}, \text{ExceptionStackFrame}). \]

lstore_<n>:

The instructions lstore_<n>, for \( 0 \leq n \leq 3 \), are typesafe iff the equivalent lstore instruction is type safe.

\[ \text{instructionHasEquivalentTypeRule}(\text{lstore}_0, \text{lstore}(0)). \]
\[ \text{instructionHasEquivalentTypeRule}(\text{lstore}_1, \text{lstore}(1)). \]
\[ \text{instructionHasEquivalentTypeRule}(\text{lstore}_2, \text{lstore}(2)). \]
\[ \text{instructionHasEquivalentTypeRule}(\text{lstore}_3, \text{lstore}(3)). \]

lsub:

An lsub instruction is type safe iff the equivalent ladd instruction is type safe.

\[ \text{instructionHasEquivalentTypeRule}(\text{lsub}, \text{ladd}). \]
lushr:

An `lushr` instruction is type safe iff the equivalent `lshl` instruction is type safe.

```
instructionHasEquivalentTypeRule(lushr, lshl).
```

lxor:

An `lxor` instruction is type safe iff the equivalent `ladd` instruction is type safe.

```
instructionHasEquivalentTypeRule(lxor, ladd).
```

monitorenter:

A `monitorenter` instruction is type safe iff one can validly pop a type matching `reference` off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(monitorenter, _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  canPop(StackFrame, [reference], NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

monitorexit:

A `monitorexit` instruction is type safe iff the equivalent `monitorenter` instruction is type safe.

```
instructionHasEquivalentTypeRule(monitorexit, monitorenter).
```
multinewarray:

A multinewarray instruction with operands CP and Dim is type safe iff CP refers to a constant pool entry denoting an array type whose dimension is greater or equal to Dim, Dim is strictly positive, and one can validly replace Dim int types on the incoming operand stack with the type denoted by CP yielding the outgoing type state.

instructionIsTypeSafe(multinewarray(CP, Dim), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-

  CP = arrayOf(_),
  classDimension(CP, Dimension),
  Dimension >= Dim,
  Dim > 0,
  /* Make a list of Dim ints */
  findall(int, between(1, Dim, _), IntList),
  validTypeTransition(Environment, IntList, CP, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).

The dimension of an array type whose component type is also an array type is 1 more than the dimension of its component type.

classDimension(arrayOf(X), Dimension) :-
  classDimension(X, Dimension1),
  Dimension is Dimension1 + 1.
classDimension(_, Dimension) :- Dimension = 0.
new:

A new instruction with operand CP at offset Offset is type safe iff CP refers to a constant pool entry denoting a class type, the type uninitialized(Offset) does not appear in the incoming operand stack, and one can validly push uninitialized(Offset) onto the incoming operand stack and replace uninitialized(Offset) with top in the incoming local variables yielding the outgoing type state.

instructionIsTypeSafe(new(CP), Environment, Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, OperandStack, Flags),
    CP = class(_, _),
   NewItem = uninitialized(Offset),
    notMemberNewItem, OperandStack),
    substituteNewItem, top, Locals, NewLocals),
    validTypeTransition(Environment, [],NewItem, frame(NewLocals, OperandStack, Flags),
    NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).

newarray:

A newarray instruction with operand TypeCode is type safe iff TypeCode corresponds to the primitive type ElementType, and one can validly replace the type int on the incoming operand stack with the type ‘array of ElementType’, yielding the outgoing type state.

instructionIsTypeSafe(newarray(TypeCode), Environment, _Offset, _StackFrame, NextStackFrame, ExceptionStackFrame) :-
    primitiveArrayInfo(TypeCode, _TypeChar, ElementType, _VerifierType),
    validTypeTransition(Environment, [], arrayOf(ElementType), _StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
The correspondence between type codes and primitive types is specified by the following predicate:

```
primitiveArrayInfo(4, 0'Z, boolean, int).
primitiveArrayInfo(5, 0'C, char, int).
primitiveArrayInfo(6, 0'F, float, float).
primitiveArrayInfo(7, 0'D, double, double).
primitiveArrayInfo(8, 0'B, byte, int).
primitiveArrayInfo(9, 0'S, short, int).
primitiveArrayInfo(10, 0'I, int, int).
primitiveArrayInfo(11, 0'J, long, long).
```

**nop:**

A *nop* instruction is always type safe. The *nop* instruction does not affect the type state.

```
instructionIsTypeSafe(nop, _Environment, _Offset, StackFrame, StackFrame, ExceptionStackFrame) :-
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**pop:**

A *pop* instruction is type safe iff one can validly pop a category 1 type off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(pop, _Environment, _Offset, StackFrame, 
    NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, [Type | Rest], Flags),
    Type \leq top,
    sizeof(Type, 1),
    NextStackFrame = frame(Locals, Rest, Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
**pop2:**

A `pop2` instruction is type safe iff it is a type safe form of the `pop2` instruction.

```prolog
instructionIsTypeSafe(pop2, _Environment, _Offset, StackFrame, 
NextStackFrame, ExceptionStackFrame) :- 
StackFrame = frame(Locals, InputOperandStack, Flags), 
pop2SomeFormIsTypeSafe(InputOperandStack, 
OutputOperandStack), 
NextStackFrame = frame(Locals, OutputOperandStack, Flags), 
exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

A `pop2` instruction is a type safe form of the `pop2` instruction iff it is a type safe form 1 `pop2` instruction or a type safe form 2 `pop2` instruction.

```prolog
pop2SomeFormIsTypeSafe(InputOperandStack, OutputOperandStack) :- 
pop2Form1IsTypeSafe(InputOperandStack, OutputOperandStack).
```

A `pop2` instruction is a type safe form 1 `pop2` instruction iff one can validly pop two types of size 1 off the incoming operand stack yielding the outgoing type state.

```prolog
pop2Form1IsTypeSafe([Type1, Type2 | Rest], Rest) :- 
  sizeOf(Type1, 1), 
  sizeOf(Type2, 1).
```

A `pop2` instruction is a type safe form 2 `pop2` instruction iff one can validly pop a type of size 2 off the incoming operand stack yielding the outgoing type state.

```prolog
pop2Form2IsTypeSafe([top, Type | Rest], Rest) :- sizeOf(Type, 2).
```
**putfield:**

A **putfield** instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is **FieldType**, declared in a class **FieldClass**, and one can validly pop types matching **FieldType** and **FieldClass** off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(putfield(CP), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    CP = field(FieldClass, FieldName, FieldDescriptor),
    parseFieldDescriptor(FieldDescriptor, FieldType),
    canPop(StackFrame, [FieldType], PoppedFrame),
    passesProtectedCheck(Environment, FieldClass, FieldName, FieldDescriptor, PoppedFrame),
    currentClassLoader(Environment, CurrentLoader),
    canPop(StackFrame, [FieldType, class(FieldClass, CurrentLoader)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

**putstatic:**

A **putstatic** instruction with operand CP is type safe iff CP refers to a constant pool entry denoting a field whose declared type is **FieldType**, and one can validly pop a type matching **FieldType** off the incoming operand stack yielding the outgoing type state.

```
instructionIsTypeSafe(putstatic(CP), _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    CP = field(_FieldClass, _FieldName, FieldDescriptor),
    parseFieldDescriptor(FieldDescriptor, FieldType),
    canPop(StackFrame, [FieldType], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
return:

A `return` instruction is type safe if the enclosing method declares a `void` return type, and either:

- The enclosing method is not an `<init>` method, or
- `this` has already been completely initialized at the point where the instruction occurs.

```
instructionIsTypeSafe(return, Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
  thisMethodReturnType(Environment, void),
  StackFrame = frame(_Locals, _OperandStack, Flags),
  notMember(flagThisUninit, Flags),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

saload:

An `saload` instruction is type safe iff one can validly replace types matching `int` and array of `short` on the incoming operand stack with `int` yielding the outgoing type state.

```
instructionIsTypeSafe(saload, Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
  validTypeTransition(Environment, [int, arrayOf(short)], int, StackFrame, NextStackFrame),
  exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
sastore:

An **sastore** instruction is type safe iff one can validly pop types matching **int**, **int** and array of **short** off the incoming operand stack yielding the outgoing type state.

```prolog
instructionIsTypeSafe(sastore, _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    canPop(StackFrame, [int, int, arrayOf(short)], NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

sipush:

An **sipush** instruction is type safe iff one can validly push the type **int** onto the incoming operand stack yielding the outgoing type state.

```prolog
instructionIsTypeSafe(sipush(_Value), Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    validTypeTransition(Environment, [], int, StackFrame, NextStackFrame),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

swap:

A **swap** instruction is type safe iff one can validly replace two category 1 types, **Type1** and **Type2**, on the incoming operand stack with the types **Type2** and **Type1** yielding the outgoing type state.

```prolog
instructionIsTypeSafe(swap, _Environment, _Offset, StackFrame, NextStackFrame, ExceptionStackFrame) :-
    StackFrame = frame(_Locals, [Type1, Type2 | Rest], _Flags),
    sizeOf(Type1, 1),
    sizeOf(Type2, 1),
    NextStackFrame = frame(_Locals, [Type2, Type1 | Rest], _Flags),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```
tables\textit{witch}:

A \textit{tableswitch} instruction is type safe if its keys are sorted, one can validly pop \texttt{int} off the incoming operand stack yielding a new type state \texttt{BranchStackFrame}, and all of the instructions targets are valid branch targets assuming \texttt{BranchStackFrame} as their incoming type state.

\begin{verbatim}
instructionIsTypeSafe(tableswitch(Targets, Keys), Environment, _Offset, StackFrame, afterGoto, ExceptionStackFrame) :-
    sort(Keys, Keys),
    canPop(StackFrame, [int], BranchStackFrame),
    checklist(targetIsTypeSafe(Environment, BranchStackFrame), Targets),
    exceptionStackFrame(StackFrame, ExceptionStackFrame).
\end{verbatim}

\textbf{wide}:

The \texttt{wide} instructions follow the same rules as the instructions they widen.

\begin{verbatim}
instructionHasEquivalentTypeRule(wide(WidenedInstruction), WidenedInstruction).
\end{verbatim}
The type state after an instruction completes abruptly is the same as the incoming type state, except that the operand stack is empty.

```
exceptionStackFrame(StackFrame, ExceptionStackFrame) :-
    StackFrame = frame(Locals, _OperandStack, Flags),
    ExceptionStackFrame = frame(Locals, [], Flags).
```

Most of the type rules in this specification depend on the notion of a valid type transition.

A type transition is valid if one can pop a list of expected types off the incoming type state’s operand stack and replace them with an expected result type, resulting in a new valid type state. In particular, the size of the operand stack in the new type state must not exceed its maximum declared size.

```
validTypeTransition(Environment, ExpectedTypesOnStack, ResultType,
    frame(Locals, InputOperandStack, Flags),
    frame(Locals, NextOperandStack, Flags)) :-
    popMatchingList(InputOperandStack, ExpectedTypesOnStack, InterimOperandStack),
    pushOperandStack(InterimOperandStack, ResultType, NextOperandStack),
    operandStackHasLegalLength(Environment, NextOperandStack).
```

Access $i$th element of the operand stack from a type state.

```
nth1OperandStackIs(I, frame(_Locals, OperandStack, _Flags),
    Element) :-
    nth1(I, OperandStack, Element).
```
4.10.2 Verification by Type Inference

A class file that does not contain a StackMapTable attribute (which necessarily has a version number of 49.0 or below) must be verified using type inference.

4.10.2.1 The Process of Verification by Type Inference

During linking, the verifier checks the code array of the Code attribute for each method of the class file by performing data-flow analysis on each method. The verifier ensures that at any given point in the program, no matter what code path is taken to reach that point, the following is true:

- The operand stack is always the same size and contains the same types of values.
- No local variable is accessed unless it is known to contain a value of an appropriate type.
- Methods are invoked with the appropriate arguments.
- Fields are assigned only using values of appropriate types.
- All opcodes have appropriate type arguments on the operand stack and in the local variable array.
- There is never an uninitialized class instance in a local variable in code protected by an exception handler. However, an uninitialized class instance may be on the operand stack in code protected by an exception handler. When an exception is thrown, the contents of the operand stack are discarded.

For details of the data-flow analysis, see Section 4.10.2.2, “The Bytecode Verifier.”

For efficiency reasons, certain tests that could in principle be performed by the verifier are delayed until the first time the code for the method is actually invoked. In so doing, the verifier avoids loading class files unless it has to.

For example, if a method invokes another method that returns an instance of class A, and that instance is assigned only to a field of the same type, the verifier does not bother to check if the class A actually exists. However, if it is assigned to a field of the type B, the definitions of both A and B must be loaded in to ensure that A is a subclass of B.
4.10.2.2 The Bytecode Verifier

This section looks at the verification of Java virtual machine code by type inference in more detail.

The code for each method is verified independently. First, the bytes that make up the code are broken up into a sequence of instructions, and the index into the code array of the start of each instruction is placed in an array. The verifier then goes through the code a second time and parses the instructions. During this pass a data structure is built to hold information about each Java virtual machine instruction in the method. The operands, if any, of each instruction are checked to make sure they are valid. For instance:

- Branches must be within the bounds of the code array for the method.

- The targets of all control-flow instructions are each the start of an instruction. In the case of a *wide* instruction, the *wide* opcode is considered the start of the instruction, and the opcode giving the operation modified by that *wide* instruction is not considered to start an instruction. Branches into the middle of an instruction are disallowed.

- No instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates.

- All references to the constant pool must be to an entry of the appropriate type. For example: the instruction *getfield* must reference a field.

- The code does not end in the middle of an instruction.

- Execution cannot fall off the end of the code.

- For each exception handler, the starting and ending point of code protected by the handler must be at the beginning of an instruction or, in the case of the ending point, immediately past the end of the code. The starting point must be before the ending point. The exception handler code must start at a valid instruction, and it must not start at an opcode being modified by the *wide* instruction.

For each instruction of the method, the verifier records the contents of the operand stack and the contents of the local variable array prior to the execution of that instruction. For the operand stack, it needs to know the stack height and the type of each value on it. For each local variable, it needs to know either the type of the contents of that local variable or that the local variable contains an unusable or
unknown value (it might be uninitialized). The bytecode verifier does not need to
distinguish between the integral types (e.g., byte, short, char) when
determining the value types on the operand stack.

Next, a data-flow analyzer is initialized. For the first instruction of the
method, the local variables that represent parameters initially contain values of the
types indicated by the method’s type descriptor; the operand stack is empty. All
other local variables contain an illegal value. For the other instructions, which
have not been examined yet, no information is available regarding the operand
stack or local variables.

Finally, the data-flow analyzer is run. For each instruction, a “changed” bit
indicates whether this instruction needs to be looked at. Initially, the “changed” bit
is set only for the first instruction. The data-flow analyzer executes the following
loop:

1. Select a virtual machine instruction whose “changed” bit is set. If no instruc-
tion remains whose “changed” bit is set, the method has successfully been ver-
ified. Otherwise, turn off the “changed” bit of the selected instruction.

2. Model the effect of the instruction on the operand stack and local variable array
by doing the following:

   • If the instruction uses values from the operand stack, ensure that
     there are a sufficient number of values on the stack and that the top
     values on the stack are of an appropriate type. Otherwise, verifica-
     tion fails.

   • If the instruction uses a local variable, ensure that the specified local
     variable contains a value of the appropriate type. Otherwise, verifi-
     cation fails.

   • If the instruction pushes values onto the operand stack, ensure that
     there is sufficient room on the operand stack for the new values. Add
     the indicated types to the top of the modeled operand stack.

   • If the instruction modifies a local variable, record that the local vari-
     able now contains the new type.

3. Determine the instructions that can follow the current instruction. Successor
instructions can be one of the following:
• The next instruction, if the current instruction is not an unconditional control transfer instruction (for instance goto, return, or athrow). Verification fails if it is possible to “fall off” the last instruction of the method.

• The target(s) of a conditional or unconditional branch or switch.

• Any exception handlers for this instruction.

4. Merge the state of the operand stack and local variable array at the end of the execution of the current instruction into each of the successor instructions. In the special case of control transfer to an exception handler, the operand stack is set to contain a single object of the exception type indicated by the exception handler information. There must be sufficient room on the operand stack for this single value, as if an instruction had pushed it.

• If this is the first time the successor instruction has been visited, record that the operand stack and local variable values calculated in steps 2 and 3 are the state of the operand stack and local variable array prior to executing the successor instruction. Set the “changed” bit for the successor instruction.

• If the successor instruction has been seen before, merge the operand stack and local variable values calculated in steps 2 and 3 into the values already there. Set the “changed” bit if there is any modification to the values.

5. Continue at step 1.

To merge two operand stacks, the number of values on each stack must be identical. The types of values on the stacks must also be identical, except that differently typed reference values may appear at corresponding places on the two stacks. In this case, the merged operand stack contains a reference to an instance of the first common superclass of the two types. Such a reference type always exists because the type Object is a superclass of all class and interface types. If the operand stacks cannot be merged, verification of the method fails.

To merge two local variable array states, corresponding pairs of local variables are compared. If the two types are not identical, then unless both contain reference values, the verifier records that the local variable contains an unusable value. If both of the pair of local variables contain reference values, the merged
state contains a reference to an instance of the first common superclass of the two types.

If the data-flow analyzer runs on a method without reporting a verification failure, then the method has been successfully verified by the class file verifier.

Certain instructions and data types complicate the data-flow analyzer. We now examine each of these in more detail.

### 4.10.2.3 Values of Types `long` and `double`

Values of the `long` and `double` types are treated specially by the verification process.

Whenever a value of type `long` or `double` is moved into a local variable at index \( n \), index \( n + 1 \) is specially marked to indicate that it has been reserved by the value at index \( n \) and must not be used as a local variable index. Any value previously at index \( n + 1 \) becomes unusable.

Whenever a value is moved to a local variable at index \( n \), the index \( n - 1 \) is examined to see if it is the index of a value of type `long` or `double`. If so, the local variable at index \( n - 1 \) is changed to indicate that it now contains an unusable value. Since the local variable at index \( n \) has been overwritten, the local variable at index \( n - 1 \) cannot represent a value of type `long` or `double`.

Dealing with values of types `long` or `double` on the operand stack is simpler; the verifier treats them as single values on the stack. For example, the verification code for the `dadd` opcode (add two `double` values) checks that the top two items on the stack are both of type `double`. When calculating operand stack length, values of type `long` and `double` have length two.

Untyped instructions that manipulate the operand stack must treat values of type `double` and `long` as atomic (indivisible). For example, the verifier reports a failure if the top value on the stack is a `double` and it encounters an instruction such as `pop` or `dup`. The instructions `pop2` or `dup2` must be used instead.

### 4.10.2.4 Instance Initialization Methods and Newly Created Objects

Creating a new class instance is a multistep process. The statement

```java
... 
new MyClass(i, j, k);
...```
can be implemented by the following:

```java
...  
new  #1  // Allocate uninitialized space for myClass
dup   // Duplicate object on the operand stack
iload_1  // Push i
iload_2  // Push j
iload_3  // Push k
invokespecial #5  // Invoke myClass.<init>
...  
```

This instruction sequence leaves the newly created and initialized object on top of the operand stack. (Additional examples of compilation to the instruction set of the Java virtual machine are given in Chapter 7, “Compiling for the Java Virtual Machine.”)

The instance initialization method (§3.9) for class myClass sees the new uninitialized object as its this argument in local variable 0. Before that method invokes another instance initialization method of myClass or its direct superclass on this, the only operation the method can perform on this is assigning fields declared within myClass.

When doing dataflow analysis on instance methods, the verifier initializes local variable 0 to contain an object of the current class, or, for instance initialization methods, local variable 0 contains a special type indicating an uninitialized object. After an appropriate instance initialization method is invoked (from the current class or the current superclass) on this object, all occurrences of this special type on the verifier’s model of the operand stack and in the local variable array are replaced by the current class type. The verifier rejects code that uses the new object before it has been initialized or that initializes the object more than once. In addition, it ensures that every normal return of the method has invoked an instance initialization method either in the class of this method or in the direct superclass.

Similarly, a special type is created and pushed on the verifier’s model of the operand stack as the result of the Java virtual machine instruction new. The special type indicates the instruction by which the class instance was created and the type of the uninitialized class instance created. When an instance initialization method is invoked on that class instance, all occurrences of the special type are replaced by the intended type of the class instance. This change in type may propagate to subsequent instructions as the dataflow analysis proceeds.
The instruction number needs to be stored as part of the special type, as there may be multiple not-yet-initialized instances of a class in existence on the operand stack at one time. For example, the Java virtual machine instruction sequence that implements

```java
new InputStream(new Foo(), new InputStream("foo"))
```

may have two uninitialized instances of InputStream on the operand stack at once. When an instance initialization method is invoked on a class instance, only those occurrences of the special type on the operand stack or in the local variable array that are the same object as the class instance are replaced.

A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a finally clause. Otherwise, a devious piece of code might fool the verifier into thinking it had initialized a class instance when it had, in fact, initialized a class instance created in a previous pass through a loop.

### 4.10.2.5 Exceptions and finally

To implement the try–finally construct, Sun’s compiler for the Java programming language (before Java SE 6) uses the exception-handling facilities together with two special instructions: `jsr` (“jump to subroutine”) and `ret` (“return from subroutine”). The finally clause is compiled as a subroutine within the Java virtual machine code for its method, much like the code for an exception handler. When a `jsr` instruction that invokes the subroutine is executed, it pushes its return address, the address of the instruction after the `jsr` that is being executed, onto the operand stack as a value of type `returnAddress`. The code for the subroutine stores the return address in a local variable. At the end of the subroutine, a `ret` instruction fetches the return address from the local variable and transfers control to the instruction at the return address.

Control can be transferred to the finally clause (the finally subroutine can be invoked) in several different ways. If the try clause completes normally, the finally subroutine is invoked via a `jsr` instruction before evaluating the next expression. A break or continue inside the try clause that transfers control outside the try clause executes a `jsr` to the code for the finally clause first. If the try clause executes a return, the compiled code does the following:

1. Saves the return value (if any) in a local variable.
2. Executes a `jsr` to the code for the finally clause.
3. Upon return from the finally clause, returns the value saved in the local variable.

The compiler sets up a special exception handler, which catches any exception thrown by the try clause. If an exception is thrown in the try clause, this exception handler does the following:

1. Saves the exception in a local variable.
2. Executes a jsr to the finally clause.
3. Upon return from the finally clause, rethrows the exception.

For more information about the implementation of the try-finally construct, see Section 7.13, “Compiling finally.”

The code for the finally clause presents a special problem to the verifier. Usually, if a particular instruction can be reached via multiple paths and a particular local variable contains incompatible values through those multiple paths, then the local variable becomes unusable. However, a finally clause might be called from several different places, yielding several different circumstances:

- The invocation from the exception handler may have a certain local variable that contains an exception.
- The invocation to implement return may have some local variable that contains the return value.
- The invocation from the bottom of the try clause may have an indeterminate value in that same local variable.

The code for the finally clause itself might pass verification, but after completing the updating all the successors of the ret instruction, the verifier would note that the local variable that the exception handler expects to hold an exception, or that the return code expects to hold a return value, now contains an indeterminate value.

Verifying code that contains a finally clause is complicated. The basic idea is the following:

- Each instruction keeps track of the list of jsr targets needed to reach that instruction. For most code, this list is empty. For instructions inside code for the finally clause, it is of length one. For multiply nested finally code (extremely rare!), it may be longer than one.
• For each instruction and each \textit{jsr} needed to reach that instruction, a bit vector is maintained of all local variables accessed or modified since the execution of the \textit{jsr} instruction.

• When executing the \textit{ret} instruction, which implements a return from a subroutine, there must be only one possible subroutine from which the instruction can be returning. Two different subroutines cannot “merge” their execution to a single \textit{ret} instruction.

• To perform the data-flow analysis on a \textit{ret} instruction, a special procedure is used. Since the verifier knows the subroutine from which the instruction must be returning, it can find all the \textit{jsr} instructions that call the subroutine and merge the state of the operand stack and local variable array at the time of the \textit{ret} instruction into the operand stack and local variable array of the instructions following the \textit{jsr}. Merging uses a special set of values for local variables:
  • For any local variable that the bit vector (constructed above) indicates has been accessed or modified by the subroutine, use the type of the local variable at the time of the \textit{ret}.
  • For other local variables, use the type of the local variable before the \textit{jsr} instruction.

\section{4.11 Limitations of the Java Virtual Machine}

The following limitations of the Java virtual machine are implicit in the \texttt{class} file format:

• The per-class or per-interface constant pool is limited to 65535 entries by the 16-bit \texttt{constant\_pool\_count} field of the \texttt{ClassFile} structure (§4.1). This acts as an internal limit on the total complexity of a single class or interface.

• The greatest number of local variables in the local variables array of a frame created upon invocation of a method is limited to 65535 by the size of the \texttt{max\_locals} item of the \texttt{Code} attribute (§4.7.3) giving the code of the method, and by the 16-bit local variable indexing of the Java virtual machine instruction set. Note that values of type \texttt{long} and \texttt{double} are each considered to reserve two local variables and contribute two units toward the \texttt{max\_locals} value, so use of local variables of those types further reduces this limit.
The number of fields that may be declared by a class or interface is limited to 65535 by the size of the fields_count item of the ClassFile structure (§4.1). Note that the value of the fields_count item of the ClassFile structure does not include fields that are inherited from superclasses or superinterfaces.

The number of methods that may be declared by a class or interface is limited to 65535 by the size of the methods_count item of the ClassFile structure (§4.1). Note that the value of the methods_count item of the ClassFile structure does not include methods that are inherited from superclasses or superinterfaces.

The number of direct superinterfaces of a class or interface is limited to 65535 by the size of the interfaces_count item of the ClassFile structure (§4.1).

The size of an operand stack in a frame (§3.6) is limited to 65535 values by the max_stack field of the Code attribute (§4.7.3). Note that values of type long and double are each considered to contribute two units toward the max_stack value, so use of values of these types on the operand stack further reduces this limit.

The number of dimensions in an array is limited to 255 by the size of the dimensions opcode of the multianewarray instruction and by the constraints imposed on the multianewarray, anewarray, and newarray instructions by §4.9.1 and §4.9.2.

The number of method parameters is limited to 255 by the definition of a method descriptor (§4.3.3), where the limit includes one unit for this in the case of instance or interface method invocations. Note that a method descriptor is defined in terms of a notion of method parameter length in which a parameter of type long or double contributes two units to the length, so parameters of these types further reduce the limit.

The length of field and method names, field and method descriptors, and other constant string values (including those referenced by ConstantValue (§4.7.2) attributes) is limited to 65535 characters by the 16-bit unsigned length item of the CONSTANT_Utf8_info structure (§4.4.7). Note that the limit is on the number of bytes in the encoding and not on the number of encoded characters. UTF-8 encodes some characters using two or three bytes. Thus, strings incorporating multibyte characters are further constrained.
Loading, Linking, and Initializing

The Java virtual machine dynamically loads, links and initializes classes and interfaces. Loading is the process of finding the binary representation of a class or interface type with a particular name and creating a class or interface from that binary representation. Linking is the process of taking a class or interface and combining it into the runtime state of the Java virtual machine so that it can be executed. Initialization of a class or interface consists of executing the class or interface initialization method <clinit> (§3.9).

In this chapter, Section 5.1 describes how the Java virtual machine derives symbolic references from the binary representation of a class or interface. Section 5.2 explains how the processes of loading, linking, and initialization are first initiated by the Java virtual machine. Section 5.3 specifies how binary representations of classes and interfaces are loaded by class loaders and how classes and interfaces are created. Linking is described in Section 5.4. Section 5.5 details how classes and interfaces are initialized. Section 5.6 introduces the notion of binding native methods. Finally, Section 5.7 describes when a Java virtual machine exits.

5.1 The Runtime Constant Pool

The Java virtual machine maintains a per-type constant pool (§3.5.5), a runtime data structure that serves many of the purposes of the symbol table of a conventional programming language implementation.

The constant_pool table (§4.4) in the binary representation of a class or interface is used to construct the runtime constant pool upon class or interface creation (§5.3). All references in the runtime constant pool are initially symbolic.
The symbolic references in the runtime constant pool are derived from structures in the binary representation of the class or interface as follows:

- A symbolic reference to a class or interface is derived from a CONSTANT_Class_info structure (§4.4.1) in the binary representation of a class or interface. Such a reference gives the name of the class or interface in the form returned by the `Class.getName` method, that is:
  - For a nonarray class or an interface, the name is the binary name (§4.2.1) of the class or interface.
  - For an array class of \( M \) dimensions, the name begins with \( M \) occurrences of the ASCII “[” character followed by a representation of the element type:
    - If the element type is a primitive type, it is represented by the corresponding field descriptor (§4.3.2).
    - Otherwise, if the element type is a reference type, it is represented by the ASCII “L” character followed by the binary name (§4.2.1) of the element type followed by the ASCII “;” character.
  - Whenever this chapter refers to the name of a class or interface, it should be understood to be in the form returned by the `Class.getName` method.

- A symbolic reference to a field of a class or an interface is derived from a CONSTANT_Fieldref_info structure (§4.4.2) in the binary representation of a class or interface. Such a reference gives the name and descriptor of the field, as well as a symbolic reference to the class or interface in which the field is to be found.

- A symbolic reference to a method of a class is derived from a CONSTANT_Methodref_info structure (§4.4.2) in the binary representation of a class or interface. Such a reference gives the name and descriptor of the method, as well as a symbolic reference to the class in which the method is to be found.

- A symbolic reference to a method of an interface is derived from a CONSTANT_InterfaceMethodref_info structure (§4.4.2) in the binary representation of a class or interface. Such a reference gives the name and descriptor of the interface method, as well as a symbolic reference to the interface in which the method is to be found.
In addition, certain non-reference runtime values are derived from items found in the \texttt{constant\_pool} table:

- A string literal (JLS3 §3.10.5) is derived from a \texttt{CONSTANT\_String\_info} structure (§4.4.3) in the binary representation of a class or interface. The \texttt{CONSTANT\_String\_info} structure gives the sequence of Unicode characters constituting the string literal.

- The Java programming language requires that identical string literals (that is, literals that contain the same sequence of characters) must refer to the same instance of class \texttt{String} (JLS3 §3.10.5). In addition, if the method \texttt{String.intern} is called on any string, the result is a reference to the same class instance that would be returned if that string appeared as a literal. Thus,

  
  
  \begin{verbatim}
  ("a" + "b" + "c").intern() == "abc"
  
  
  \end{verbatim}

  must have the value true.

- To derive a string literal, the Java virtual machine examines the sequence of characters given by the \texttt{CONSTANT\_String\_info} structure.

- If the method \texttt{String.intern} has previously been called on an instance of class \texttt{String} containing a sequence of Unicode characters identical to that given by the \texttt{CONSTANT\_String\_info} structure, then the result of string literal derivation is a reference to that same instance of class \texttt{String}.

- Otherwise, a new instance of class \texttt{String} is created containing the sequence of Unicode characters given by the \texttt{CONSTANT\_String\_info} structure; that class instance is the result of string literal derivation. Finally, the \texttt{intern} method of the new \texttt{String} instance is invoked.

- Runtime constant values are derived from \texttt{CONSTANT\_Integer\_info}, \texttt{CONSTANT\_Float\_info}, \texttt{CONSTANT\_Long\_info}, or \texttt{CONSTANT\_Double\_info} structures (§4.4.4, §4.4.5) in the binary representation of a class or interface. Note that \texttt{CONSTANT\_Float\_info} structures represent values in IEEE 754 single format and \texttt{CONSTANT\_Double\_info} structures represent values in IEEE 754 double format (§4.4.4, §4.4.5). The runtime constant values derived from these structures must thus be values that can be represented using IEEE 754 single and double formats, respectively.

  The remaining structures in the \texttt{constant\_pool} table of the binary representation of a class or interface, the \texttt{CONSTANT\_NameAndType\_info} (§4.4.6) and
CONSTANT_Utf8_info (§4.4.7) structures are only used indirectly when deriving symbolic references to classes, interfaces, methods, and fields, and when deriving string literals.

5.2 Virtual Machine Start-up

The Java virtual machine starts up by creating an initial class, which is specified in an implementation-dependent manner, using the bootstrap class loader (§5.3.1). The Java virtual machine then links the initial class, initializes it, and invokes the public class method `void main(String[])`. The invocation of this method drives all further execution. Execution of the Java virtual machine instructions constituting the `main` method may cause linking (and consequently creation) of additional classes and interfaces, as well as invocation of additional methods.

In an implementation of the Java virtual machine, the initial class could be provided as a command line argument. Alternatively, the implementation could provide an initial class that sets up a class loader which in turn loads an application. Other choices of the initial class are possible so long as they are consistent with the specification given in the previous paragraph.

5.3 Creation and Loading

Creation of a class or interface `C` denoted by the name `N` consists of the construction in the method area of the Java virtual machine (§3.5.4) of an implementation-specific internal representation of `C`. Class or interface creation is triggered by another class or interface `D`, which references `C` through its runtime constant pool. Class or interface creation may also be triggered by `D` invoking methods in certain Java class libraries (§3.12) such as reflection.

If `C` is not an array class, it is created by loading a binary representation of `C` (see Chapter 4, “The class File Format”) using a class loader (JLS3 §12.2). Array classes do not have an external binary representation; they are created by the Java virtual machine rather than by a class loader.

There are two kinds of class loaders: the bootstrap class loader supplied by the Java virtual machine, and user-defined class loaders. Every user-defined class loader is an instance of a subclass of the abstract class `ClassLoader`. Applications employ user-defined class loaders in order to extend the manner in which the Java virtual machine dynamically loads and thereby creates classes. User-defined class
loaders can be used to create classes that originate from user-defined sources. For example, a class could be downloaded across a network, generated on the fly, or extracted from an encrypted file. If a user-defined class loader prefetches binary representations of classes and interfaces, or loads a group of related classes together, then it must reflect loading errors only at points in the program where they could have arisen without prefetching or group loading.

A class loader \( L \) may create \( C \) by defining it directly or by delegating to another class loader. If \( L \) creates \( C \) directly, we say that \( L \) defines \( C \) or, equivalently, that \( L \) is the defining loader of \( C \).

When one class loader delegates to another class loader, the loader that initiates the loading is not necessarily the same loader that completes the loading and defines the class. If \( L \) creates \( C \), either by defining it directly or by delegation, we say that \( L \) initiates loading of \( C \) or, equivalently, that \( L \) is an initiating loader of \( C \).

At runtime, a class or interface is determined not by its name alone, but by a pair: its binary name (§4.2.1) and its defining class loader. Each such class or interface belongs to a single runtime package. The runtime package of a class or interface is determined by the package name and defining class loader of the class or interface.

The Java virtual machine uses one of three procedures to create class or interface \( C \) denoted by \( N \):

- If \( N \) denotes a nonarray class or an interface, one of the two following methods is used to load and thereby create \( C \):
  - If \( D \) was defined by the bootstrap class loader, then the bootstrap class loader initiates loading of \( C \) (§5.3.1).
  - If \( D \) was defined by a user-defined class loader, then that same user-defined class loader initiates loading of \( C \) (§5.3.2).
- Otherwise \( N \) denotes an array class. An array class is created directly by the Java virtual machine (§5.3.3), not by a class loader. However, the defining class loader of \( D \) is used in the process of creating array class \( C \).

If an error occurs during class loading, then an instance of a subclass of LinkageError must be thrown at a point in the program that (directly or indirectly) uses the class or interface being loaded.

We will sometimes represent a class or interface using the notation \(<N, Ld>\), where \( N \) denotes the name of the class or interface and \( Ld \) denotes the defining loader of the class or interface. We will also represent a class or interface using the
notation $N_{Li}$, where $N$ denotes the name of the class or interface and $Li$ denotes an initiating loader of the class or interface.

### 5.3.1 Loading Using the Bootstrap Class Loader

The following steps are used to load and thereby create the nonarray class or interface $C$ denoted by $N$ using the bootstrap class loader.

First, the Java virtual machine determines whether the bootstrap class loader has already been recorded as an initiating loader of a class or interface denoted by $N$. If so, this class or interface is $C$, and no class creation is necessary.

Otherwise, the Java virtual machine performs one of the following two operations in order to load $C$:

1. The Java virtual machine searches for a purported representation of $C$ in a platform-dependent manner. Typically, a class or interface will be represented using a file in a hierarchical file system, and the name of the class or interface will be encoded in the pathname of the file. Note that there is no guarantee that a purported representation found is valid or is a representation of $C$.

   This phase of loading must detect the following error:

   - If no purported representation of $C$ is found, loading throws an instance of `NoClassDefFoundError` or an instance of one of its subclasses.

   Then the Java virtual machine attempts to derive a class denoted by $N$ using the bootstrap class loader from the purported representation using the algorithm found in Section 5.3.5. That class is $C$.

2. The bootstrap class loader can delegate the loading of $C$ to some user-defined class loader $L$ by passing $N$ to an invocation of a `loadClass` method on $L$. The result of the invocation is $C$. The Java virtual machine then records that the bootstrap loader is an initiating loader of $C$ (§5.3.4).

### 5.3.2 Loading Using a User-defined Class Loader

The following steps are used to load and thereby create the nonarray class or interface $C$ denoted by $N$ using a user-defined class loader $L$.

First, the Java virtual machine determines whether $L$ has already been recorded as an initiating loader of a class or interface denoted by $N$. If so, this class or interface is $C$, and no class creation is necessary.
Otherwise the Java virtual machine invokes `loadClass(N)` on `L`.\(^1\) The value returned by the invocation is the created class or interface `C`. The Java virtual machine then records that `L` is an initiating loader of `C` (§5.3.4). The remainder of this section describes this process in more detail.

When the `loadClass` method of the class loader `L` is invoked with the name `N` of a class or interface `C` to be loaded, `L` must perform one of the following two operations in order to load `C`:

1. The class loader `L` can create an array of bytes representing `C` as the bytes of a `ClassFile` structure (§4.1); it then must invoke the method `defineClass` of class `ClassLoader`. Invoking `defineClass` causes the Java virtual machine to derive a class or interface denoted by `N` using `L` from the array of bytes using the algorithm found in Section 5.3.5.

2. The class loader `L` can delegate the loading of `C` to some other class loader `L'`. This is accomplished by passing the argument `N` directly or indirectly to an invocation of a method on `L'` (typically the `loadClass` method). The result of the invocation is `C`.

### 5.3.3 Creating Array Classes

The following steps are used to create the array class `C` denoted by `N` using class loader `L`. Class loader `L` may be either the bootstrap class loader or a user-defined class loader.

If `L` has already been recorded as an initiating loader of an array class with the same component type as `N`, that class is `C`, and no array class creation is necessary. Otherwise, the following steps are performed to create `C`:

1. If the component type is a reference type, the algorithm of this section (§5.3) is applied recursively using class loader `L` in order to load and thereby create the component type of `C`.

---

\(^1\) Since JDK release 1.1, Sun’s Java virtual machine implementation has invoked the `loadClass` method of a class loader in order to cause it to load a class or interface. The argument to `loadClass` is the name of the class or interface to be loaded. There is also a two-argument version of the `loadClass` method, where the second argument is a `boolean` that indicates whether the class or interface is to be linked or not. Only the two-argument version was supplied in JDK release 1.0.2, and Sun’s Java virtual machine implementation relied on it to link the loaded class or interface. From JDK release 1.1 onward, Sun’s Java virtual machine implementation links the class or interface directly, without relying on the class loader.
2. The Java virtual machine creates a new array class with the indicated component type and number of dimensions. If the component type is a reference type, \( C \) is marked as having been defined by the defining class loader of the component type. Otherwise, \( C \) is marked as having been defined by the bootstrap class loader. In any case, the Java virtual machine then records that \( L \) is an initiating loader for \( C \) (§5.3.4). If the component type is a reference type, the accessibility of the array class is determined by the accessibility of its component type. Otherwise, the accessibility of the array class is public.

### 5.3.4 Loading Constraints

Ensuring type safe linkage in the presence of class loaders requires special care. It is possible that when two different class loaders initiate loading of a class or interface denoted by \( N \), the name \( N \) may denote a different class or interface in each loader.

When a class or interface \( C = <N_1, L_1> \) makes a symbolic reference to a field or method of another class or interface \( D = <N_2, L_2> \), the symbolic reference includes a descriptor specifying the type of the field, or the return and argument types of the method. It is essential that any type name \( N \) mentioned in the field or method descriptor denote the same class or interface when loaded by \( L_1 \) and when loaded by \( L_2 \).

To ensure this, the Java virtual machine imposes **loading constraints** of the form \( N^{L_1} = N^{L_2} \) during preparation (§5.4.2) and resolution (§5.4.3). To enforce these constraints, the Java virtual machine will, at certain prescribed times (see §5.3.1, §5.3.2, §5.3.3, and §5.3.5), record that a particular loader is an initiating loader of a particular class. After recording that a loader is an initiating loader of a class, the Java virtual machine must immediately check to see if any loading constraints are violated. If so, the record is retracted, the Java virtual machine throws a **LinkageError**, and the loading operation that caused the recording to take place fails.

Similarly, after imposing a loading constraint (see §5.4.2, §5.4.3.3, §5.4.3.4, and §5.4.3.5), the Java virtual machine must immediately check to see if any loading constraints are violated. If so, the newly imposed loading constraint is retracted, the Java virtual machine throws a **LinkageError**, and the operation that caused the constraint to be imposed (either resolution or preparation, as the case may be) fails.

The situations described here are the only times at which the Java virtual machine checks whether any loading constraints have been violated. A loading constraint is **violated** if, and only if, all the following four conditions hold:
• There exists a loader $L$ such that $L$ has been recorded by the Java virtual machine as an initiating loader of a class $C$ named $N$.

• There exists a loader $L'$ such that $L'$ has been recorded by the Java virtual machine as an initiating loader of a class $C'$ named $N$.

• The equivalence relation defined by the (transitive closure of the) set of imposed constraints implies $N^L = N^{L'}$.

• $C \neq C'$.

A full discussion of class loaders and type safety is beyond the scope of this specification. For a more comprehensive discussion, readers are referred to Dynamic Class Loading in the Java Virtual Machine by Sheng Liang and Gilad Bracha (Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications).

5.3.5 Deriving a Class from a class File Representation

The following steps are used to derive a Class object for the nonarray class or interface $C$ denoted by $N$ using loader $L$ from a purported representation in class file format.

1. First, the Java virtual machine determines whether it has already recorded that $L$ is an initiating loader of a class or interface denoted by $N$. If so, this creation attempt is invalid and loading throws a LinkageError.

2. Otherwise, the Java virtual machine attempts to parse the purported representation. However, the purported representation may not in fact be a valid representation of $C$.

   This phase of loading must detect the following errors:

   • If the purported representation is not a ClassFile structure ($\S 4.1, \S 4.8$), loading throws an instance of ClassFormatError.

   • Otherwise, if the purported representation is not of a supported major or minor version ($\S 4.1$), loading throws an instance of UnsupportedClassVersionError.

   • Otherwise, if the purported representation does not actually represent a class named $N$, loading throws an instance of NoClassDefFoundError or an instance of one of its subclasses.
3. If \( C \) has a direct superclass, the symbolic reference from \( C \) to its direct superclass is resolved using the algorithm of Section 5.4.3.1. Note that if \( C \) is an interface it must have \texttt{Object} as its direct superclass, which must already have been loaded. Only \texttt{Object} has no direct superclass.

Any exceptions that can be thrown due to class or interface resolution can be thrown as a result of this phase of loading. In addition, this phase of loading must detect the following errors:

- If the class or interface named as the direct superclass of \( C \) is in fact an interface, loading throws an \texttt{IncompatibleClassChangeError}.

- Otherwise, if any of the superclasses of \( C \) is \( C \) itself, loading throws a \texttt{ClassCircularityError}.

4. If \( C \) has any direct superinterfaces, the symbolic references from \( C \) to its direct superinterfaces are resolved using the algorithm of Section 5.4.3.1.

Any exceptions that can be thrown due to class or interface resolution can be thrown as a result of this phase of loading. In addition, this phase of loading must detect the following errors:

- If any of the classes or interfaces named as direct superinterfaces of \( C \) is not in fact an interface, loading throws an \texttt{IncompatibleClassChangeError}.

- Otherwise, if any of the superinterfaces of \( C \) is \( C \) itself, loading throws a \texttt{ClassCircularityError}.

5. The Java virtual machine marks \( C \) as having \( L \) as its defining class loader and records that \( L \) is an initiating loader of \( C \) (§5.3.4).

---

1 \texttt{UnsupportedClassVersionError}, a subclass of \texttt{ClassFormatError}, was introduced to enable easy identification of a \texttt{ClassFormatError} caused by an attempt to load a class whose representation uses an unsupported version of the class file format. In JDK release 1.1 and earlier, an instance of \texttt{NoClassDefFoundError} or \texttt{ClassFormatError} was thrown in case of an unsupported version, depending on whether the class was being loaded by the system class loader or a user-defined class loader.
5.4 Linking

Linking a class or interface involves verifying and preparing that class or interface, its direct superclass, its direct superinterfaces, and its element type (if it is an array type), if necessary. Resolution of symbolic references in the class or interface is an optional part of linking.

This specification allows an implementation flexibility as to when linking activities (and, because of recursion, loading) take place, provided that all of the following properties are maintained:

- A class or interface is completely loaded before it is linked.
- A class or interface is completely verified and prepared before it is initialized.
- Errors detected during linkage are thrown at a point in the program where some action is taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error.

For example, a Java virtual machine implementation may choose to resolve each symbolic reference in a class or interface individually when it is used (“lazy” or “late” resolution), or to resolve them all at once when the class is being verified (“eager” or “static” resolution). This means that the resolution process may continue, in some implementations, after a class or interface has been initialized. Whichever strategy is followed, any error detected during resolution must be thrown at a point in the program that (directly or indirectly) uses a symbolic reference to the class or interface.

Because linking involves the allocation of new data structures, it may fail with an OutOfMemoryError.

5.4.1 Verification

Verification (§4.10) ensures that the binary representation of a class or interface is structurally correct (§4.9). Verification may cause additional classes and interfaces to be loaded (§5.3) but need not cause them to be verified or prepared.

If the binary representation of a class or interface does not satisfy the static or structural constraints listed in Section 4.9, “Constraints on Java Virtual Machine Code,” then a VerifyError must be thrown at the point in the program that caused the class or interface to be verified.

If an attempt by the Java virtual machine to verify a class or interface fails because an error is thrown that is an instance of LinkageError (or a subclass),
then subsequent attempts to verify the class or interface always fail with the same error that was thrown as a result of the initial verification attempt.

5.4.2 Preparation

Preparation involves creating the static fields for a class or interface and initializing such fields to their default values (JLS3 §4.12.5). This does not require the execution of any Java virtual machine code; explicit initializers for static fields are executed as part of initialization (§5.5), not preparation.

During preparation of a class or interface $C$, the Java virtual machine also imposes loading constraints (§5.3.4). Let $L_1$ be the defining loader of $C$. For each method $m$ declared in $C$ that overrides (§5.4.2.1) a method declared in a superclass or superinterface $<D, L_2>$, the Java virtual machine imposes the following loading constraints: Let $T_0$ be the name of the type returned by $m$, and let $T_1$, ..., $T_n$ be the names of the argument types of $m$. Then $T_i^{L_1} = T_i^{L_2}$ for $i = 0$ to $n$ (§5.3.4).

Furthermore, if $C$ implements a method $m$ declared in a superinterface $<I, L_3>$ of $C$, but $C$ does not itself declare the method $m$, then let $<D, L_2>$, be the superclass of $C$ that declares the implementation of method $m$ inherited by $C$. The Java virtual machine imposes the following constraints:

Let $T_0$ be the name of the type returned by $m$, and let $T_1$, ..., $T_n$ be the names of the argument types of $m$. Then $T_i^{L_2} = T_i^{L_3}$ for $i = 0$ to $n$ (§5.3.4).

Preparation may occur at any time following creation but must be completed prior to initialization.

5.4.2.1 Method overriding

An instance method $m_1$ declared in class $C$ overrides another instance method $m_2$ declared in class $A$ iff all of the following are true:

- $C$ is a subclass of $A$.
- $m_2$ has the same name and descriptor as $m_1$.
- Either:
  - $m_2$ is marked ACC_PUBLIC; or is marked ACC_PROTECTED; or is marked neither ACC_PUBLIC nor ACC_PROTECTED nor ACC_PRIVATE and belongs to the same runtime package as $C$, or
• \(m1\) overrides a method \(m3\), \(m3\) distinct from \(m1\), \(m3\) distinct from \(m2\), such that \(m3\) overrides \(m2\).

5.4.3 Resolution

Resolution is the process of dynamically determining concrete values from symbolic references in the runtime constant pool.

Resolution can be attempted on a symbolic reference that has already been resolved. An attempt to resolve a symbolic reference that has already successfully been resolved always succeeds trivially and always results in the same entity produced by the initial resolution of that reference.

If an error occurs during resolution of a symbolic reference, then an instance of IncompatibleClassChangeError (or a subclass) must be thrown at a point in the program that (directly or indirectly) uses the symbolic reference.

If an attempt by the Java virtual machine to resolve a symbolic reference fails because an error is thrown that is an instance of LinkageError (or a subclass), then subsequent attempts to resolve the reference always fail with the same error that was thrown as a result of the initial resolution attempt.

The Java virtual machine instructions anewarray, checkcast, getfield, getstatic, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, multianewarray, new, putfield, and putstatic make symbolic references to the runtime constant pool. Execution of any of these instructions requires resolution of its symbolic reference.

Certain of these instructions require additional linking checks when resolving symbolic references. For instance, in order for a getfield instruction to successfully resolve the symbolic reference to the field on which it operates, it must not only complete the field resolution steps given in Section 5.4.3.3 but also check that the field is not static. If it is a static field, a linking exception must be thrown.

Linking exceptions generated by checks that are specific to the execution of a particular Java virtual machine instruction are given in the description of that instruction and are not covered in this general discussion of resolution. Note that such exceptions, although described as part of the execution of Java virtual machine instructions rather than resolution, are still properly considered failures of resolution.

The following sections describe the process of resolving a symbolic reference in the runtime constant pool (§5.1) of a class or interface \(D\). Details of resolution differ with the kind of symbolic reference to be resolved.
5.4.3.2 Class and Interface Resolution

To resolve an unresolved symbolic reference from $D$ to a class or interface $C$ denoted by $N$, the following steps are performed:

1. The defining class loader of $D$ is used to create a class or interface denoted by $N$. This class or interface is $C$. Any exception that can be thrown as a result of failure of class or interface creation can thus be thrown as a result of failure of class and interface resolution. The details of the process are given in Section 5.3.

2. If $C$ is an array class and its element type is a reference type, then the symbolic reference to the class or interface representing the element type is resolved by invoking the algorithm in Section 5.4.3.1 recursively.

3. Finally, access permissions to $C$ are checked:
   - If $C$ is not accessible (§5.4.4) to $D$, class or interface resolution throws an `IllegalAccessError`.

   This condition can occur, for example, if $C$ is a class that was originally declared to be `public` but was changed to be non-`public` after $D$ was compiled.

If steps 1 and 2 succeed but step 3 fails, $C$ is still valid and usable. Nevertheless, resolution fails, and $D$ is prohibited from accessing $C$.

5.4.3.3 Field Resolution

To resolve an unresolved symbolic reference from $D$ to a field in a class or interface $C$, the symbolic reference to $C$ given by the field reference must first be resolved (§5.4.3.1). Therefore, any exception that can be thrown as a result of failure of resolution of a class or interface reference can be thrown as a result of failure of field resolution. If the reference to $C$ can be successfully resolved, an exception relating to the failure of resolution of the field reference itself can be thrown.

When resolving a field reference, field resolution first attempts to look up the referenced field in $C$ and its superclasses:

1. If $C$ declares a field with the name and descriptor specified by the field reference, field lookup succeeds. The declared field is the result of the field lookup.

2. Otherwise, field lookup is applied recursively to the direct superinterfaces of the specified class or interface $C$.

3. Otherwise, if $C$ has a superclass $S$, field lookup is applied recursively to $S$. 
4. Otherwise, field lookup fails.

   If field lookup fails, field resolution throws a NoSuchFieldError. Otherwise, if field lookup succeeds but the referenced field is not accessible (§5.4.4) to D, field resolution throws an IllegalAccessError.

   Otherwise, let \(<E, L1>\) be the class or interface in which the referenced field is actually declared and let \(L2\) be the defining loader of \(D\). Let \(T\) be the name of the type of the referenced field. The Java virtual machine must impose the loading constraint that \(T^{L1} = T^{L2}\) (§5.3.4).

5.4.3.4 Method Resolution

To resolve an unresolved symbolic reference from \(D\) to a method in a class \(C\), the symbolic reference to \(C\) given by the method reference is first resolved (§5.4.3.1). Therefore, any exceptions that can be thrown due to resolution of a class reference can be thrown as a result of method resolution. If the reference to \(C\) can be successfully resolved, exceptions relating to the resolution of the method reference itself can be thrown.

   When resolving a method reference:

1. Method resolution checks whether \(C\) is a class or an interface.

   - If \(C\) is an interface, method resolution throws an IncompatibleClassChangeError.

2. Method resolution attempts to look up the referenced method in \(C\) and its superclasses:

   - If \(C\) declares a method with the name and descriptor specified by the method reference, method lookup succeeds.

   - Otherwise, if \(C\) has a superclass, step 2 of method lookup is recursively invoked on the direct superclass of \(C\).

3. Otherwise, method lookup attempts to locate the referenced method in any of the superinterfaces of the specified class \(C\).

   - If any superinterface of \(C\) declares a method with the name and descriptor specified by the method reference, method lookup succeeds.

   - Otherwise, method lookup fails.

   If method lookup fails, method resolution throws a NoSuchMethodError. If method lookup succeeds and the method is abstract, but \(C\) is not abstract,
method resolution throws an `AbstractMethodError`. Otherwise, if the referenced method is not accessible (§5.4.4) to `D`, method resolution throws an `IllegalAccessError`.

Otherwise, let `<E, L1>` be the class or interface in which the referenced method is actually declared and let `L2` be the defining loader of `D`. Let `T0` be the name of the type returned by the referenced method, and let `T1, ..., Tn` be the names of the argument types of the referenced method. The Java virtual machine must impose the loading constraints `Ti^{L1} = Ti^{L2}` for `i = 0` to `n` (§5.3.4).

**5.4.3.5 Interface Method Resolution**

To resolve an unresolved symbolic reference from `D` to an interface method in an interface `C`, the symbolic reference to `C` given by the interface method reference is first resolved (§5.4.3.1). Therefore, any exceptions that can be thrown as a result of failure of resolution of an interface reference can be thrown as a result of failure of interface method resolution. If the reference to `C` can be successfully resolved, exceptions relating to the resolution of the interface method reference itself can be thrown.

When resolving an interface method reference:

- If `C` is not an interface, interface method resolution throws an `IncompatibleClassChangeError`.

- Otherwise, if the referenced method does not have the same name and descriptor as a method in `C` or in one of the superinterfaces of `C`, or in class `Object`, interface method resolution throws a `NoSuchMethodError`.

  Otherwise, let `<E, L1>` be the class or interface in which the referenced interface method is actually declared and let `L2` be the defining loader of `D`. Let `T0` be the name of the type returned by the referenced method, and let `T1, ..., Tn` be the names of the argument types of the referenced method. The Java virtual machine must impose the loading constraints `Ti^{L1} = Ti^{L2}` for `i = 0` to `n` (§5.3.4).

**5.4.4 Access Control**

A class or interface `C` is *accessible* to a class or interface `D` if and only if either of the following conditions are true:

- `C` is public.

- `C` and `D` are members of the same runtime package (§5.3).
A field or method $R$ is accessible to a class or interface $D$ if and only if any of the following conditions is true:

- $R$ is public.

- $R$ is protected and is declared in a class $C$, and $D$ is either a subclass of $C$ or $C$ itself. Furthermore, if $R$ is not static, then the symbolic reference to $R$ must contain a symbolic reference to a class $T$, such that $T$ is either a subclass of $D$, a superclass of $D$ or $D$ itself.

- $R$ is either protected or has default access (that is, neither public nor protected nor private), and is declared by a class in the same runtime package as $D$.

- $R$ is private and is declared in $D$.

This discussion of access control omits a related restriction on the target of a protected field access or method invocation (the target must be of class $D$ or a subtype of $D$). That requirement is checked as part of the verification process (§5.4.1); it is not part of link-time access control.

### 5.5 Initialization

*Initialization* of a class or interface consists of executing its class or interface initialization method (§3.9). A class or interface may be initialized only as a result of:

- The execution of any one of the Java virtual machine instructions `new`, `getstatic`, `putstatic`, or `invokestatic` that references the class or interface. All of these instructions reference a class directly or indirectly through either a field reference or a method reference. Upon execution of a `new` instruction, the referenced class or interface is initialized if it has not been initialized already. Upon execution of a `getstatic`, `putstatic`, or `invokestatic` instruction, the class or interface that declared the resolved field or method is initialized if it has not been initialized already.

- Invocation of certain reflective methods in the class library (§3.12), for example, in class `Class` or in package `java.lang.reflect`.

- The initialization of one of its subclasses.

- Its designation as the initial class at Java virtual machine start-up (§5.2).
Prior to initialization, a class or interface must be linked, that is, verified, prepared, and optionally resolved.

Because the Java virtual machine is multithreaded, initialization of a class or interface requires careful synchronization, since some other thread may be trying to initialize the same class or interface at the same time. There is also the possibility that initialization of a class or interface may be requested recursively as part of the initialization of that class or interface. The implementation of the Java virtual machine is responsible for taking care of synchronization and recursive initialization by using the following procedure. It assumes that the Class object has already been verified and prepared, and that the Class object contains state that indicates one of four situations:

- This Class object is verified and prepared but not initialized.
- This Class object is being initialized by some particular thread.
- This Class object is fully initialized and ready for use.
- This Class object is in an erroneous state, perhaps because initialization was attempted and failed.

For each class or interface $C$, there is a unique initialization lock $LC$. The mapping from $C$ to $LC$ is left to the discretion of the Java virtual machine implementation. The procedure for initializing $C$ is then as follows:

1. Synchronize on the initialization lock, $LC$, for $C$. This involves waiting until the current thread can acquire $LC$.
2. If the Class object for $C$ indicates that initialization is in progress for $C$ by some other thread, then release LC and block the current thread until informed that the in-progress initialization has completed, at which time repeat this step.
3. If the Class object for $C$ indicates that initialization is in progress for $C$ by the current thread, then this must be a recursive request for initialization. Release $LC$ and complete normally.
4. If the Class object for $C$ indicates that $C$ has already been initialized, then no further action is required. Release $LC$ and complete normally.
5. If the Class object for $C$ is in an erroneous state, then initialization is not possible. Release $LC$ and throw a NoClassDefFoundError.
6. Otherwise, record the fact that initialization of the Class object for C is in progress by the current thread, and release LC.

7. Next, if C is a class rather than an interface, and its superclass SC has not yet been initialized, then recursively perform this entire procedure for SC. If necessary, verify and prepare SC first. If the initialization of SC completes abruptly because of a thrown exception, then acquire LC, label the Class object for C as erroneous, notify all waiting threads, release LC, and complete abruptly, throwing the same exception that resulted from initializing SC.

8. Next, determine whether assertions are enabled for C by querying its defining class loader.

9. Next, execute the class or interface initialization method of C.

10. If the execution of the class or interface initialization method completes normally, then acquire LC, label the Class object for C as fully initialized, notify all waiting threads, release LC, and complete this procedure normally.

11. Otherwise, the class or interface initialization method must have completed abruptly by throwing some exception \( E \). If the class of \( E \) is not Error or one of its subclasses, then create a new instance of the class ExceptionInInitializerError with \( E \) as the argument, and use this object in place of \( E \) in the following step. If a new instance of ExceptionInInitializerError cannot be created because an OutOfMemoryError occurs, then use an OutOfMemoryError object in place of \( E \) in the following step.

12. Acquire LC, label the Class object for C as erroneous, notify all waiting threads, release LC, and complete this procedure abruptly with reason \( E \) or its replacement as determined in the previous step.

An implementation may optimize this procedure by eliding the lock acquisition in step 1 (and release in step 4/5) when it can determine that the initialization of the class has already completed, provided that, in terms of the memory model, all happens-before orderings (JLS §17.4.5) that would exist if the lock were acquired, still exist when the optimization is performed.

5.6 Binding Native Method Implementations

*Binding* is the process by which a function written in a language other than the Java programming language and implementing a native method is integrated into the Java virtual machine so that it can be executed. Although this process is traditionally
referred to as linking, the term binding is used in the specification to avoid confusion with linking of classes or interfaces by the Java virtual machine.

### 5.7 Virtual Machine Exit

The Java virtual machine terminates all its activity and exits when either:

- All threads that are not daemon threads terminate.
- Some thread invokes the `exit` method of class `Runtime` or class `System`, and the `exit` operation is permitted by the security manager.
A Java 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 virtual machine instruction and the operation it performs.

6.1 Assumptions: The Meaning of “Must”

The description of each instruction is always given in the context of Java virtual machine code that satisfies the static and structural constraints of Chapter 4, “The class File Format.” In the description of individual Java 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 4 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 runtime, the behavior of the Java virtual machine is undefined.

The Java virtual machine checks that Java virtual machine code satisfies the static and structural constraints at link time using a class file verifier (see Section 4.10, “Verification of class Files”). Thus, a Java virtual machine will only attempt to execute code from valid class files. Performing verification at link time is attractive in that the checks are performed just once, substantially reducing the amount of work that must be done at runtime. Other implementation strategies are possible, provided that they comply with The Java Language Specification and The Java Virtual Machine Specification.
6.2 Reserved Opcodes

In addition to the opcodes of the instructions specified later in this chapter, which are used in class files (see Chapter 4, “The class File Format”), three opcodes are reserved for internal use by a Java virtual machine implementation. If Sun extends the instruction set of the Java virtual machine in the future, these reserved opcodes are guaranteed not to be used.

Two of the 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. The third reserved opcode, number 202 (0xca), has the mnemonic breakpoint and is intended to be used by debuggers to implement breakpoints.

Although these opcodes have been reserved, they may be used only inside a Java virtual machine implementation. They cannot appear in valid class files. Tools such as debuggers or JIT code generators (§3.13) that might directly interact with Java virtual machine code that has been already loaded and executed may encounter these opcodes. Such tools should attempt to behave gracefully if they encounter any of these reserved instructions.

6.3 Virtual Machine Errors

A Java virtual machine implementation throws an object that is an instance of a subclass of the class VirtualMachineError when an internal error or resource limitation prevents it from implementing the semantics of the Java programming language. This specification cannot predict where internal errors or resource limitations may be encountered and does not mandate precisely when they can be reported. Thus, any of the VirtualMachineError subclasses defined below may be thrown at any time during the operation of the Java virtual machine:

- InternalError: An internal error has occurred in the Java virtual machine implementation because of a fault in the software implementing the virtual machine, a fault in the underlying host system software, or a fault in the hardware. This error is delivered asynchronously (§3.10) when it is detected and may occur at any point in a program.
- **OutOfMemoryError**: The Java virtual machine implementation has run out of either virtual or physical memory, and the automatic storage manager was unable to reclaim enough memory to satisfy an object creation request.

- **StackOverflowError**: The Java virtual machine implementation has run out of stack space for a thread, typically because the thread is doing an unbounded number of recursive invocations as a result of a fault in the executing program.

- **UnknownError**: An exception or error has occurred, but the Java virtual machine implementation is unable to report the actual exception or error.

### 6.4 Format of Instruction Descriptions

Java virtual machine instructions are represented in this chapter by entries of the form shown in Figure 6.1, in alphabetical order and each beginning on a new page.

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

Keep in mind that there are “operands” generated at compile time and embedded within Java virtual machine instructions, as well as “operands” calculated at runtime 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 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 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 `lconst_<l>` family of instructions, giving mnemonic and opcode information for the two instructions in that family (`lconst_0` and `lconst_1`), is

**Forms**

\[
\begin{align*}
lconst_0 &= 9 \ (0x9) \\
lconst_1 &= 10 \ (0xa)
\end{align*}
\]
<table>
<thead>
<tr>
<th><strong>mnemonic</strong></th>
<th><strong>mnemonic</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Operation</strong></th>
<th>Short description of the instruction</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Format</strong></th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>mnemonic</code></td>
</tr>
<tr>
<td></td>
<td><code>operand1</code></td>
</tr>
<tr>
<td></td>
<td><code>operand2</code></td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Operand</strong></th>
<th><code>...</code>, <code>value1</code>, <code>value2</code> ⇒</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack</strong></td>
<td><code>...</code>, <code>value3</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th>A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed, the type of the results, etc.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Linking Exceptions</strong></th>
<th>If any linking exceptions may be thrown by the execution of this instruction, they are set off one to a line, in the order in which they must be thrown.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Runtime Exceptions</strong></th>
<th>If any runtime exceptions can be thrown by the execution of an instruction, they are set off one to a line, in the order in which they must be thrown.</th>
</tr>
</thead>
</table>

Other than the linking and runtime exceptions, if any, listed for an instruction, that instruction must not throw any runtime exceptions except for instances of `VirtualMachineError` or its subclasses.

<table>
<thead>
<tr>
<th><strong>Notes</strong></th>
<th>Comments not strictly part of the specification of an instruction are set aside as notes at the end of the description.</th>
</tr>
</thead>
</table>

Figure 6.1 An example instruction page
In the description of the Java virtual machine instructions, the effect of an instruction’s execution on the operand stack (§3.6.2) of the current frame (§3.6) is represented textually, with the stack growing from left to right and each value represented separately. Thus,

Operand \( \ldots, \text{value1}, \text{value2} \Rightarrow \)

Stack \( \ldots, \text{result} \)

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

Values of types long and double are represented by a single entry on the operand stack.\(^1\)

---

\(^1\) In the first edition of this specification, values on the operand stack of types long and double were each represented in the stack diagram by two entries.
**aaload**

**Operation**  Load reference from array

**Format**

```
 aaload
```

**Forms**

\[ aaload = 50 \ (0x32) \]

**Operand Stack**

\[ ..., \text{arrayref}, \text{index} \Rightarrow \]

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

**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 `int`. 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 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`.

**aaload**

```
 aaload
```

**Operation**  Load reference from array

**Format**

```
 aaload
```

**Forms**

\[ aaload = 50 \ (0x32) \]

**Operand Stack**

\[ ..., \text{arrayref}, \text{index} \Rightarrow \]

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

**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 `int`. 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 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`.
**aastore**

**Operation**  
Store into reference array

**Format**

```
aastore
```

**Forms**

```
aastore = 83 (0x53)
```

**Operand 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 `int` and `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`.

At runtime, the type of `value` must be compatible with the type of the components of the array referenced by `arrayref`. Specifically, assignment of a value of reference type `S` (source) to an array component of reference type `T` (target) is allowed only if:

- 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, `S` must implement interface `T`.
  
- If `S` is an interface type, then:
  - If `T` is a class type, then `T` must be `Object`.
  - If `T` is an interface type, then `T` must be the same interface as `S` or a superinterface of `S`. 

---

**aastore**
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 $TC[]$, that is, an array of components of type $TC$, then one of the following must be true:
  - $TC$ and $SC$ are the same primitive type.
  - $TC$ and $SC$ are reference types, and type $SC$ is assignable to $TC$ by these runtime rules.

- If $T$ is an interface type, $T$ must be one of the interfaces implemented by arrays (JLS3 §4.10.3).

### Runtime Exceptions

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

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

Otherwise, if $arrayref$ is not null and the actual type of $value$ is not assignment compatible (JLS3 §5.2) with the actual type of the components of the array, $aastore$ throws an `ArrayStoreException`.
\textit{aconst\_null} \hspace{1cm} \textit{aconst\_null}

\textbf{Operation} \hspace{1cm} \text{Push null}

\textbf{Format} \hspace{1cm} \textit{aconst\_null}

\textbf{Forms} \hspace{1cm} \textit{aconst\_null} = 1 (0x1)

\textbf{Operand} \hspace{1cm} \ldots \Rightarrow

\textbf{Stack} \hspace{1cm} \ldots, null

\textbf{Description} \hspace{1cm} \text{Push the null object reference onto the operand stack.}

\textbf{Notes} \hspace{1cm} \text{The Java virtual machine does not mandate a concrete value for null.}
**aload**

**Operation**
Load reference from local variable

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aload</code></td>
<td><code>index</code></td>
</tr>
</tbody>
</table>

**Forms**
`aload = 25 (0x19)`

**Operand Stack**
`... ⇒ ..., objectref`

**Description**
The `index` is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). 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 `return-Address` from a local variable onto the operand stack. This asymmetry with the `astore` instruction is intentional.

The `aload` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
**aload\_<n>**

**Operation**  Load reference from local variable

**Format**  

```
aload\_<n>
```

**Forms**  

- `aload_0 = 42 (0x2a)`
- `aload_1 = 43 (0x2b)`
- `aload_2 = 44 (0x2c)`
- `aload_3 = 45 (0x2d)`

**Operand Stack**  

... ⇒ ...

..., objectref

**Description**  The `<n>` must be an index into the local variable array of the current frame (§3.6). The local variable at `<n>` must contain a **reference**. The **objectref** in the local variable at index 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.
**anewarray**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Create new array of reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td><img src="#" alt="Format Table" /></td>
</tr>
<tr>
<td>Forms</td>
<td>$\text{anewarray} = 189$ (0xbd)</td>
</tr>
<tr>
<td>Operand</td>
<td>$\ldots, \text{count} \Rightarrow$</td>
</tr>
<tr>
<td>Stack</td>
<td>$\ldots, \text{arrayref}$</td>
</tr>
<tr>
<td>Description</td>
<td>The <code>count</code> must be of type <code>int</code>. It is popped off the operand stack. The <code>count</code> represents the number of components of the array to be created. The unsigned <code>indexbyte1</code> and <code>indexbyte2</code> are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is $(\text{indexbyte1} \ll 8) \mid \text{indexbyte2}$. The runtime constant pool item at that index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). A new array with components of that type, of length <code>count</code>, is allocated from the garbage-collected heap, and a reference <code>arrayref</code> to this new array object is pushed onto the operand stack. All components of the new array are initialized to <code>null</code>, the default value for reference types (JLS3 §4.12.5).</td>
</tr>
<tr>
<td>Linking Exceptions</td>
<td>During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.</td>
</tr>
<tr>
<td>Runtime Exception</td>
<td>Otherwise, if <code>count</code> is less than zero, the <code>anewarray</code> instruction throws a <code>NegativeArraySizeException</code>.</td>
</tr>
<tr>
<td>Notes</td>
<td>The <code>anewarray</code> instruction is used to create a single dimension of an array of object references or part of a multidimensional array.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>anewarray</code></td>
</tr>
<tr>
<td><code>indexbyte1</code></td>
</tr>
<tr>
<td><code>indexbyte2</code></td>
</tr>
</tbody>
</table>
areturn

Operation
Return reference from method

Format

Forms
areturn = 176 (0xb0)

Operand
..., objectref ⇒

Stack
[empty]

Description
The objectref must be of type reference and must refer to an object of a type that is assignment compatible (JLS3 §5.2) with the type represented by the return descriptor (§4.3.3) of the current method. If the current method is a synchronized method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a monitorexit instruction in the current thread. If no exception is thrown, objectref is popped from the operand stack of the current frame (§3.6) 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 interpreter then reinstates the frame of the invoker and returns control to the invoker.

Runtime Exceptions
If the virtual machine implementation does not enforce the rules on structured locking described in §3.11.10, then if the current method is a synchronized method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, areturn throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monotorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the first of those rules is violated during invocation of the current method, then areturn throws an IllegalMonitorStateException.
arraylength

Operation  Get length of array

Format  

Forms  arraylength = 190 (0xbe)

Operand  ..., arrayref ⇒

Stack  ..., 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 operand stack as an int.

Runtime Exception  If the arrayref is null, the arraylength instruction throws a NullPointerException.
**astore**

**Operation**  Store reference into local variable

**Format**

```
| astore | index |
```

**Forms**  
`astore = 58 (0x3a)`

**Operand Stack**  
... , `objectref` ⇒

**Description**  
The `index` is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). 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 `index` is set to `objectref`.

**Notes**  
The `astore` instruction is used with an `objectref` of type `returnAddress` when implementing the `finally` clause of the Java programming language (see §7.13, “Compiling finally”). 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.

The `astore` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
\textbf{astore_\(<n>\)}

**Operation**
Store reference into local variable

**Format**

\[ \text{astore_\(n\)} \]

**Forms**
- \text{astore\_0} = 75 (0x4b)
- \text{astore\_1} = 76 (0x4c)
- \text{astore\_2} = 77 (0x4d)
- \text{astore\_3} = 78 (0x4e)

**Operand Stack**
- \(\ldots, \text{oobjectref} \Rightarrow \)
- \(\ldots\)

**Description**
The \(<n>\) must be an index into the local variable array of the current frame (§3.6). The \text{objectref} on the top of the operand stack must be of type \text{returnAddress} or of type reference. It is popped from the operand stack, and the value of the local variable at \(<n>\) is set to \text{objectref}.

**Notes**
An \text{astore_\(<n>\)} instruction is used with an \text{objectref} of type \text{returnAddress} when implementing the \textit{finally} clauses of the Java programming language (see §7.13, “Compiling \textit{finally}”). An \text{aload_\(<n>\)} instruction cannot be used to load a value of type \text{returnAddress} from a local variable onto the operand stack. This asymmetry with the corresponding \text{astore_\(<n>\)} instruction is intentional.

Each of the \text{astore_\(<n>\)} instructions is the same as \text{astore} with an \textit{index} of \(<n>\), except that the operand \(<n>\) is implicit.
**athrow**

**Operation**  
Throw exception or error

**Format**  

```
athrow
```

**Forms**  
```
athrow = 191 (0xbf)
```

**Operand Stack**  
```
..., objectref ⇒
```

**Stack**  
```
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 method (§3.6) for the first exception handler that matches the class of `objectref`, as given by the algorithm in §3.10.

If an exception handler that matches `objectref` 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 matching exception handler is found in the current frame, that frame is popped. If the current frame represents an invocation of a synchronized method, the monitor entered or reentered on invocation of the method is exited as if by execution of a `monitorexit` instruction. Finally, the frame of its invoker is reinstated, if such a frame exists, and the `objectref` is rethrown. If no such frame exists, the current thread exits.

**Runtime Exceptions**  
If `objectref` is `null`, `athrow` throws a `NullPointerException` instead of `objectref`. 

Otherwise, if the virtual machine implementation does not enforce the rules on structured locking described in §3.11.10, then if the method of the current frame is a synchronized method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, `athrow` throws an `IllegalMonitorStateException` instead of the object previously being thrown. This can happen, for example, if an abruptly completing synchronized method contains a `monitorexit` instruction, but no `monitorenter` instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the first of those rules is violated during invocation of the current method, then `athrow` throws an `IllegalMonitorStateException` instead of the object previously being thrown.

**Notes**

The operand stack diagram for the `athrow` instruction may be misleading: If a handler for this exception is matched in the current method, the `athrow` instruction discards all the values on the operand stack, then pushes the thrown object onto the operand stack. However, if no handler is matched in the current method and the exception is thrown farther up the method invocation chain, then the operand stack of the method (if any) that handles the exception is cleared and `objectref` is pushed onto that empty operand stack. All intervening frames from the method that threw the exception up to, but not including, the method that handles the exception are discarded.
The Java Virtual Machine Instruction Set

**baload**

**Operation**
Load byte or boolean from array

**Format**

```
baload
```

**Forms**

```
baload = 51 (0x33)
```

**Operand 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 `int`. 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 an `int` `value`, and pushed onto the top of the operand stack.

**Runtime Exceptions**

If `arrayref` is null, `baload` throws a `NullPointerException`.

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

**Notes**
The `baload` instruction is used to load values from both byte and boolean arrays. In Sun’s implementation of the Java virtual machine, boolean arrays (arrays of type `T_BOOLEAN`; see §3.2 and the description of the `newarray` instruction in this chapter) are implemented as arrays of 8-bit values. Other implementations may implement packed boolean arrays; the `baload` instruction of such implementations must be used to access those arrays.
**Operation**  
Store into byte or boolean array

**Format**  

```
    bastore
```

**Forms**  
`bastore = 84 (0x54)`

**Operand 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 the `value` must both be of type `int`. The `arrayref`, `index`, and `value` are popped from the operand stack. The `int value` is truncated to a byte and stored as the component of the array indexed by `index`.

**Runtime Exceptions**  
If `arrayref` is `null`, `bastore` throws a `NullPointerException`. Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `bastore` instruction throws an `ArrayIndexOutOfBoundsException`.

**Notes**  
The `bastore` instruction is used to store values into both byte and boolean arrays. In Sun’s implementation of the Java virtual machine, boolean arrays (arrays of type `T_BOOLEAN`; see §3.2 and the description of the `newarray` instruction in this chapter) are implemented as arrays of 8-bit values. Other implementations may implement packed boolean arrays; in such implementations the `bastore` instruction must be able to store boolean values into packed boolean arrays as well as byte values into byte arrays.
bipush

Operation Push byte

Format

Form

Operand Stack

Description The immediate byte is sign-extended to an int value. That value is pushed onto the operand stack.
**caload**

**Operation**
Load char from array

**Format**
caload

**Forms**
caload = 52 (0x34)

**Operand Stack**
..., arrayref, index ⇒

**Stack**
..., value

**Description**
The arrayref must be of type reference and must refer to an array whose components are of type char. The index must be of type int. Both arrayref and index are popped from the operand stack. The component of the array at index is retrieved and zero-extended to an int value. That value is pushed onto the operand stack.

**Runtime Exceptions**
If arrayref is null, caload throws a NullPointerException.
Otherwise, if index is not within the bounds of the array referenced by arrayref, the caload instruction throws an ArrayIndexOutOfBoundsException.
**castore**

**Operation**  
Store into char array

**Format**  

```
castore
```

**Forms**  
```
castore = 85 (0x55)
```

**Operand**  
```
..., arrayref, index, value ⇒
```

**Stack**  
```
...
```

**Description**  
The `arrayref` must be of type `reference` and must refer to an array whose components are of type `char`. The `index` and the `value` must both be of type `int`. The `arrayref`, `index`, and `value` are popped from the operand stack. The `int value` is truncated to a `char` and stored as the component of the array indexed by `index`.

**Runtime Exceptions**  
If `arrayref` is `null`, `castore` throws a `NullPointerException`. Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `castore` instruction throws an `ArrayIndexOutOfBoundsException`.
checkcast

**Operation**
Check whether object is of given type

**Format**

<table>
<thead>
<tr>
<th>checkcast</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**

`checkcast = 192 (0xc0)`

**Operand**

`. . . objectref ⇒`

**Stack**

`. . . objectref`

**Description**
The `objectref` must be of type `reference`. The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type.

If `objectref` is `null`, the `checkcast` instruction throws a `ClassCastException`.

Otherwise, the named class, array, or interface type is resolved (§5.4.3.1). If `objectref` can be cast to the resolved class, array, 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 an ordinary (nonarray) class, then:
  - If $T$ is a class type, then $S$ must be the same class as $T$, or a subclass of $T$.
  - If $T$ is an interface type, then $S$ must implement interface $T$. 
checkcast (cont.)

- If \( S \) is an interface type, then:
  - If \( T \) is a class type, then \( T \) must be \texttt{Object}.
  - If \( T \) is an interface type, then \( T \) must be the same interface as \( S \) or a superinterface of \( S \).

- If \( S \) is a class representing the array type \( SC[] \), that is, an array of components of type \( SC \), then:
  - If \( T \) is a class type, then \( T \) must be \texttt{Object}.
  - If \( T \) is an array type \( TC[] \), that is, an array of components of type \( TC \), then one of the following must be true:
    - \( TC \) and \( SC \) are the same primitive type.
    - \( TC \) and \( SC \) are reference types, and type \( SC \) can be cast to \( TC \) by recursive application of these rules.
  - If \( T \) is an interface type, \( T \) must be one of the interfaces implemented by arrays (JLS3 §4.10.3).

Linking Exceptions

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

Runtime Exception

Otherwise, if \texttt{objectref} cannot be cast to the resolved class, array, or interface type, the \texttt{checkcast} instruction throws a \texttt{ClassCastException}.

Notes

The \texttt{checkcast} instruction is very similar to the \texttt{instanceof} instruction. It differs in its treatment of \texttt{null}, its behavior when its test fails (\texttt{checkcast} throws an exception, \texttt{instanceof} pushes a result code), and its effect on the operand stack.
**Operation**
Convert `double` to `float`

**Format**
```
d2f
```

**Forms**
`d2f = 144 (0x90)`

**Operand**
`...`, `value` ⇒

**Stack**
`...`, `result`

**Description**
The `value` on the top of the operand stack must be of type `double`. It is popped from the operand stack and undergoes value set conversion (§3.8.3) resulting in `value'`. Then `value'` is converted to a `float` `result` using IEEE 754 round to nearest mode. The `result` is pushed onto the operand stack.

Where an `d2f` instruction is FP-strict (§3.8.2), the result of the conversion is always rounded to the nearest representable value in the float value set (§3.3.2).

Where an `d2f` instruction is not FP-strict, the result of the conversion may be taken from the float-extended-exponent value set (§3.3.2); it is not necessarily rounded to the nearest representable value in the float value set.

A finite `value'` too small to be represented as a `float` is converted to a zero of the same sign; a finite `value'` too large to be represented as a `float` is converted to an infinity of the same sign. A `double` NaN is converted to a `float` NaN.

**Notes**
The `d2f` instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of `value'` and may also lose precision.
**d2i**

**Operation** Convert double to int

**Format**

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Format" /></td>
</tr>
</tbody>
</table>

**Forms**

\[d2i = 142 \ (0x8e)\]

**Operand** 

\[\ldots, \text{value} \Rightarrow\]

**Stack** 

\[\ldots, \text{result}\]

**Description** The value on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3) resulting in value'. Then value' is converted to an int. The result is pushed onto the operand stack:

- If the value' is NaN, the result of the conversion is an int 0.
- Otherwise, if the value' is not an infinity, it is rounded to an integer value V, rounding towards zero using IEEE 754 round towards zero mode. If this integer value V can be represented as an int, then the result is the int value V.

- Otherwise, either the value' must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type int, or the value' must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type int.

**Notes** The d2i instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of value' and may also lose precision.
**d2l**

**Operation** Convert double to long

**Format**

\[ d2l \]

**Forms**
\[ d2l = 143 \text{ (0x8f)} \]

**Operand** \( ..., \text{value} \Rightarrow \)

**Stack** \( ..., \text{result} \)

**Description** The value on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3) resulting in value'. Then value' is converted to a long. The result is pushed onto the operand stack:

- If the value' is NaN, the result of the conversion is a long 0.

- Otherwise, if the value' is not an infinity, it is rounded to an integer value \( V \), rounding towards zero using IEEE 754 round towards zero mode. If this integer value \( V \) can be represented as a long, then the result is the long value \( V \).

- Otherwise, either the value' must be too small (a negative value of large magnitude or negative infinity), and the result is the smallest representable value of type long, or the value' must be too large (a positive value of large magnitude or positive infinity), and the result is the largest representable value of type long.

**Notes** The d2l instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of value' and may also lose precision.
**dadd**

**Operation**  Add double

**Format**  

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dadd</strong></td>
</tr>
</tbody>
</table>

**Forms**  

$dadd = 99$ (0x63)

**Operand Stack**  

$\ldots, value1, value2 \Rightarrow$

**Stack**  

$\ldots, result$

**Description**  

Both $value1$ and $value2$ must be of type `double`. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in $value1'$ and $value2'$. The `double` result is $value1' + value2'$. The result is pushed onto the operand stack.

The result of a `dadd` instruction is governed by the rules of IEEE arithmetic:

- If either $value1'$ or $value2'$ is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
dadd (cont.)

- In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a double, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a double, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a dadd instruction never throws a runtime exception.
**daload**

**Operation**  Load double from array

**Format**  

```
daload
```

**Forms**  

\[ \text{daload} = 49 \ (0x31) \]

**Operand Stack**  

```
..., arrayref, index ⇒
```

**Description**  

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

**Runtime Exceptions**  

If `arrayref` is `null`, `daload` throws a `NullPointerException`. Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `daload` instruction throws an `ArrayIndexOutOfBoundsException`.
**dastore**

**Operation**  Store into double array

**Format**

```
\[ \text{dastore} \]
```

**Forms**  
\[ \text{dastore} = 82 \ (0x52) \]

**Operand Stack**  
\[ \ldots, \text{arrayref, index, value} \Rightarrow \]

**Description**  The `arrayref` must be of type `reference` and must refer to an array whose components are of type `double`. The `index` must be of type `int`, and `value` must be of type `double`. The `arrayref`, `index`, and `value` are popped from the operand stack. The `double` `value` undergoes value set conversion (§3.8.3), resulting in `value'`, which is stored as the component of the array indexed by `index`.

**Runtime Exceptions**

If `arrayref` is `null`, `dastore` throws a `NullPointerException`.

Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `dastore` instruction throws an `ArrayIndexOutOfBoundsException`. 
The Java Virtual Machine Instruction Set

\textit{dcmp<op>}

\textbf{Operation} \hspace{1cm} \text{Compare double}

\textbf{Format} \hspace{1cm} \textit{dcmp<op>}

\textbf{Forms} \hspace{1cm} \textit{dcmpg} = 152 (0x98)  \hspace{1cm} \textit{dcmpl} = 151 (0x97)

\textbf{Operand Stack} \hspace{1cm} \ldots, \text{value1, value2} \Rightarrow \ldots, \text{result}

\textbf{Description} \hspace{1cm} \text{Both value1 and value2 must be of type double. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1' and value2'. A floating-point comparison is performed:}

\begin{itemize}
  \item If value1' is greater than value2', the int value 1 is pushed onto the operand stack.
  \item Otherwise, if value1' is equal to value2', the int value 0 is pushed onto the operand stack.
  \item Otherwise, if value1' is less than value2', the int value \(-1\) is pushed onto the operand stack.
  \item Otherwise, at least one of value1' or value2' is NaN. The \textit{dcmpg} instruction pushes the int value 1 onto the operand stack and the \textit{dcmpl} instruction pushes the int value \(-1\) onto the operand stack.
\end{itemize}

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.
Notes

The `dcmpg` and `dcmpl` instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any double comparison fails if either or both of its operands are NaN. With both `dcmpg` and `dcmpl` available, any double comparison may be compiled to push the same result onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see §7.5, “More Control Examples.”
### dconst_<d>  \[ \text{dconst}_{<d>} \]

**Operation** Push double

**Format**

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>dconst_&lt;d&gt;</td>
</tr>
</tbody>
</table>

**Forms**

- \( dconst_0 = 14 \) (0xe)
- \( dconst_1 = 15 \) (0xf)

**Operand** … ⇒

**Stack** …, <d>

**Description** Push the double constant <d> (0.0 or 1.0) onto the operand stack.
**ddiv**

**Operation**  Divide double

**Format**  

```

```

**Forms**  

`ddiv = 111 (0x6f)`

**Operand Stack**  

`..., value1, value2 ⇒`  

**Stack**  

`..., result`

**Description**  

Both `value1` and `value2` must be of type `double`. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in `value1′` and `value2′`. The `double` result is `value1′ / value2′`. The result is pushed onto the operand stack.

The result of a `ddiv` instruction is governed by the rules of IEEE arithmetic:

- If either `value1′` or `value2′` is NaN, the result is NaN.
- If neither `value1′` nor `value2′` is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.
- Division of an infinity by an infinity results in NaN.
- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.
- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.
- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.
In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest `double` using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a `double`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `double`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of a `ddiv` instruction never throws a runtime exception.
**dload**

**Operation**  Load `double` from local variable

**Format**

```
+dload
index
```

**Forms**  `dload = 24 (0x18)`

**Operand Stack**  `… ⇒ …, value`

**Description**  The `index` is an unsigned byte. Both `index` and `index + 1` must be indices into the local variable array of the current frame (§3.6). The local variable at `index` must contain a `double`. The `value` of the local variable at `index` is pushed onto the operand stack.

**Notes**  The `dload` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
**dload_<n>**

**Operation**  Load double from local variable

**Format**  

<table>
<thead>
<tr>
<th>Format</th>
<th>dload_&lt;n&gt;</th>
</tr>
</thead>
</table>

**Forms**  

- `dload_0 = 38 (0x26)`
- `dload_1 = 39 (0x27)`
- `dload_2 = 40 (0x28)`
- `dload_3 = 41 (0x29)`

**Operand**

<table>
<thead>
<tr>
<th>Operand</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>... ⇒</td>
<td>..., value</td>
</tr>
</tbody>
</table>

**Description**  Both <n> and <n> + 1 must be indices into the local variable array of the current frame (§3.6). The local variable at <n> must contain a double. The value of the local variable at <n> is pushed onto the operand stack.

**Notes**  Each of the `dload_<n>` instructions is the same as `dload` with an index of <n>, except that the operand <n> is implicit.
\textbf{dmul} \hspace{1cm} \textbf{dmul}

\textbf{Operation} \hspace{0.5cm} Multiply double

\textbf{Format} \hspace{0.5cm} \texttt{dmul}

\textbf{Forms} \hspace{0.5cm} \texttt{dmul} = 107 (0x6b)

\textbf{Operand} \hspace{0.5cm} \ldots, \texttt{value1}, \texttt{value2} \Rightarrow

\textbf{Stack} \hspace{0.5cm} \ldots, \texttt{result}

\textbf{Description} \hspace{0.5cm} Both \texttt{value1} and \texttt{value2} must be of type double. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \texttt{value1'} and \texttt{value2'}. The double result is \texttt{value1'} * \texttt{value2'}. The result is pushed onto the operand stack.

The result of a \texttt{dmul} instruction is governed by the rules of IEEE arithmetic:

- If either \texttt{value1'} or \texttt{value2'} is NaN, the result is NaN.

- If neither \texttt{value1'} nor \texttt{value2'} is NaN, the sign of the result is positive if both values have the same sign and negative if the values have different signs.

- Multiplication of an infinity by a zero results in NaN.

- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.

- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a double, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a double, we say the operation underflows; the result is then a zero of appropriate sign.
The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a `dmul` instruction never throws a runtime exception.
**dneg**

**Operation**  Negate double

**Format**

| dneg |

**Forms**  \[ dneg = 119 \ (0x77) \]

**Operand**  \[ ..., value \Rightarrow \]

**Stack**  \[ ..., result \]

**Description**  The value must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value’. The double result is the arithmetic negation of value’. The result is pushed onto the operand stack.

For double values, negation is not the same as subtraction from zero. If \( x \) is +0.0, then 0.0–\( x \) equals +0.0, but −\( x \) equals −0.0. Unary minus merely inverts the sign of a double.

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.
The result of a `drem` instruction is governed by these rules:

- If either `value1'` or `value2'` is NaN, the result is NaN.
- If neither `value1'` nor `value2'` is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
**drem (cont.)**

- If the dividend is a zero and the divisor is finite, the result equals the dividend.

- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder result from a dividend \( value1' \) and a divisor \( value2' \) is defined by the mathematical relation \( result = value1' - (value2' \times q) \), where \( q \) is an integer that is negative only if \( value1' / value2' \) is negative, and positive only if \( value1' / value2' \) is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of \( value1' \) and \( value2' \).

Despite the fact that division by zero may occur, evaluation of a drem instruction never throws a runtime exception. Overflow, underflow, or loss of precision cannot occur.

**Notes**

The IEEE 754 remainder operation may be computed by the library routine `Math.IEEEremainder`.
**dreturn**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Return double from method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td><strong>dreturn</strong></td>
</tr>
<tr>
<td>Forms</td>
<td><strong>dreturn</strong> = 175 (0xaf)</td>
</tr>
<tr>
<td>Operand</td>
<td>..., value ⇒</td>
</tr>
<tr>
<td>Stack</td>
<td>[empty]</td>
</tr>
</tbody>
</table>

**Description**
The current method must have return type double. The value must be of type double. If the current method is a synchronized method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a monitorexit instruction in the current thread. If no exception is thrown, value is popped from the operand stack of the current frame (§3.6) and undergoes value set conversion (§3.8.3), resulting in value'. The value' is 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 interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**
If the virtual machine implementation does not enforce the rules on structured locking described in §3.11.10, then if the current method is a synchronized method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, dreturn throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the first of those rules is violated during invocation of the current method, then dreturn throws an IllegalMonitorStateException.
### dstore

**Operation**  
Store double into local variable

**Format**

<table>
<thead>
<tr>
<th>dstore</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
</tr>
</tbody>
</table>

**Forms**  
dstore = 57 (0x39)

**Operand Stack**

..., value ⇒

**Description**  
The index is an unsigned byte. Both index and index + 1 must be indices into the local variable array of the current frame (§3.6). The value on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value’. The local variables at index and index + 1 are set to value’.

**Notes**  
The dstore opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
**dstore_<n>**

**Operation** Store double into local variable

**Format**

<table>
<thead>
<tr>
<th>dstore_&lt;n&gt;</th>
</tr>
</thead>
</table>

**Forms**

- `dstore_0 = 71 (0x47)`
- `dstore_1 = 72 (0x48)`
- `dstore_2 = 73 (0x49)`
- `dstore_3 = 74 (0x4a)`

**Operand**

... , value ⇒

**Stack**

...

**Description** Both `<n>` and `<n> + 1` must be indices into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type double. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in `value'`. The local variables at `<n>` and `<n> + 1` are set to `value'`.

**Notes** Each of the `dstore_<n>` instructions is the same as `dstore` with an index of `<n>`, except that the operand `<n>` is implicit.
\textit{dsub} \quad \textit{dsub}

\begin{tabular}{|l|}
\hline
\textbf{Operation} & Subtract \texttt{double} \\
\hline
\textbf{Format} & \texttt{dsub} \\
\hline
\textbf{Forms} & \texttt{dsub} = 103 (0x67) \\
\hline
\textbf{Operand Stack} & \ldots, \textit{value1}, \textit{value2} ⇒ \\
\hline
\textbf{Description} & Both \textit{value1} and \textit{value2} must be of type \texttt{double}. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \textit{value1'} and \textit{value2'}. The \texttt{double} result is \textit{value1'} − \textit{value2'}. The \textit{result} is pushed onto the operand stack. \\
\hline
For \texttt{double} subtraction, it is always the case that \texttt{a−b} produces the same result as \texttt{a+(−b)}. However, for the \texttt{dsub} instruction, subtraction from zero is not the same as negation, because if \textit{x} is \texttt{+0.0}, then \texttt{0.0−x} equals \texttt{+0.0}, but \texttt{−x} equals \texttt{−0.0}. \\
\hline
The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of a \texttt{dsub} instruction never throws a runtime exception. \\
\hline
\end{tabular}
### dup

**Operation**  
Duplicate the top operand stack value

**Format**  
```text
dup
```

**Forms**  
`dup = 89 (0x59)`

**Operand**  
`..., value ⇒`

**Stack**  
`..., value, value`

**Description**  
Duplicate the top value on the operand stack and push the duplicated value onto the operand stack.

The `dup` instruction must not be used unless `value` is a value of a category 1 computational type (§3.11.1).
**dup_x1**

**Operation**  Duplicate the top operand stack value and insert two values down

**Format**  

```
dup_x1
```

**Forms**  

\[ \text{dup}_x1 = 90 \ (0x5a) \]

**Operand**  

\( \ldots, \text{value}_2, \text{value}_1 \Rightarrow \)

**Stack**  

\( \ldots, \text{value}_1, \text{value}_2, \text{value}_1 \)

**Description**  

Duplicate the top value on the operand stack and insert the duplicated value two values down in the operand stack.

The \textit{dup}_x1 instruction must not be used unless both \textit{value}_1 and \textit{value}_2 are values of a category 1 computational type (§3.11.1).
**dup_x2**

**Operation**  Duplicate the top operand stack value and insert two or three values down

**Format**  

```
dup_x2
```

**Forms**  

`dup_x2 = 91 (0x5b)`

**Operand Stack**

Form 1:

```
..., value3, value2, value1 ⇒
..., value1, value3, value2, value1
```

where `value1`, `value2`, and `value3` are all values of a category 1 computational type (§3.11.1).

Form 2:

```
..., value2, value1 ⇒
..., value1, value2, value1
```

where `value1` is a value of a category 1 computational type and `value2` is a value of a category 2 computational type (§3.11.1).

**Description**  Duplicate the top value on the operand stack and insert the duplicated value two or three values down in the operand stack.
**dup2**

**Operation**  Duplicate the top one or two operand stack values

**Format**  

```
dup2
```

**Forms**  \( dup2 = 92 \) (0x5c)

**Operand Stack**

Form 1:

\[ \ldots, \text{value2}, \text{value1} \Rightarrow \ldots, \text{value2}, \text{value1}, \text{value2}, \text{value1} \]

where both \( \text{value1} \) and \( \text{value2} \) are values of a category 1 computational type (§3.11.1).

Form 2:

\[ \ldots, \text{value} \Rightarrow \ldots, \text{value}, \text{value} \]

where \( \text{value} \) is a value of a category 2 computational type (§3.11.1).

**Description**  Duplicate the top one or two values on the operand stack and push the duplicated value or values back onto the operand stack in the original order.
**dup2_x1**

**Operation**  
Duplicate the top one or two operand stack values and insert two or three values down

**Format**

```
  dup2_x1
```

**Forms**  
```
dup2_x1 = 93 (0x5d)
```

**Operand Stack**

Form 1:

```
..., value3, value2, value1 ⇒
..., value2, value1, value3, value2, value1
```

where `value1`, `value2`, and `value3` are all values of a category 1 computational type (§3.11.1).

Form 2:

```
..., value2, value1 ⇒
..., value1, value2, value1
```

where `value1` is a value of a category 2 computational type and `value2` is a value of a category 1 computational type (§3.11.1).

**Description**  
Duplicate the top one or two values on the operand stack and insert the duplicated values, in the original order, one value beneath the original value or values in the operand stack.
**dup2_x2**

**Operation**  
Duplicate the top one or two operand stack values and insert two, three, or four values down

**Format**  
```
dup2_x2
```

**Forms**  
\( \text{dup2}_x2 = 94 \ (0x5e) \)

**Operand Stack**

**Form 1:**
```
..., value4, value3, value2, value1 ⇒
..., value2, value1, value4, value3, value2, value1
```

where `value1`, `value2`, `value3`, and `value4` are all values of a category 1 computational type (§3.11.1).

**Form 2:**
```
..., value3, value2, value1 ⇒
..., value1, value3, value2, value1
```

where `value1` is a value of a category 2 computational type and `value2` and `value3` are both values of a category 1 computational type (§3.11.1).

**Form 3:**
```
..., value3, value2, value1 ⇒
..., value2, value1, value3, value2, value1
```

where `value1` and `value2` are both values of a category 1 computational type and `value3` is a value of a category 2 computational type (§3.11.1).

**Form 4:**
```
..., value2, value1 ⇒
..., value1, value2, value1
```

where `value1` and `value2` are both values of a category 2 computational type (§3.11.1).
dup2_x2 (cont.)

Description  Duplicate the top one or two values on the operand stack and insert the duplicated values, in the original order, into the operand stack.
**Operation** Convert float to double

**Format**
```
f2d
```

**Forms**
f2d = 141 (0x8d)

**Operand Stack**
```
..., value ⇒ ...
```

**Stack**
```
..., result
```

**Description** The value on the top of the operand stack must be of type float. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value’. Then value’ is converted to a double result. This result is pushed onto the operand stack.

**Notes** Where an f2d instruction is FP-strict (§3.8.2) it performs a widening primitive conversion (JLS3 §5.1.2). Because all values of the float value set (§3.3.2) are exactly representable by values of the double value set (§3.3.2), such a conversion is exact.

Where an f2d instruction is not FP-strict, the result of the conversion may be taken from the double-extended-exponent value set; it is not necessarily rounded to the nearest representable value in the double value set. However, if the operand value is taken from the float-extended-exponent value set and the target result is constrained to the double value set, rounding of value may be required.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Convert float to int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td><code>f2i</code></td>
</tr>
<tr>
<td>Forms</td>
<td><code>f2i = 139 (0x8b)</code></td>
</tr>
<tr>
<td>Operand</td>
<td><code>...</code>, <code>value</code> ⇒</td>
</tr>
<tr>
<td>Stack</td>
<td><code>...</code>, <code>result</code></td>
</tr>
</tbody>
</table>

**Description**

The value on the top of the operand stack must be of type `float`. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in `value'`. Then `value'` is converted to an `int` `result`. This `result` is pushed onto the operand stack:

- If the `value'` is NaN, the `result` of the conversion is an `int 0`.
- Otherwise, if the `value'` is not an infinity, it is rounded to an integer value `V`, rounding towards zero using IEEE 754 round towards zero mode. If this integer value `V` can be represented as an `int`, then the `result` is the `int` value `V`.
- Otherwise, either the `value'` must be too small (a negative value of large magnitude or negative infinity), and the `result` is the smallest representable value of type `int`, or the `value'` must be too large (a positive value of large magnitude or positive infinity), and the `result` is the largest representable value of type `int`.

**Notes**

The `f2i` instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of `value'` and may also lose precision.
**Operation**  Convert `float` to `long`  

**Format**  

```
f2l
```

**Forms**  

`f2l = 140 (0x8c)`  

**Operand**  

`..., value ⇒`  

**Stack**  

`..., result`  

**Description**  
The `value` on the top of the operand stack must be of type `float`. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in `value'`. Then `value'` is converted to a `long result`. This `result` is pushed onto the operand stack:

- If the `value'` is NaN, the `result` of the conversion is a `long 0`.
- Otherwise, if the `value'` is not an infinity, it is rounded to an integer value $V$, rounding towards zero using IEEE 754 round towards zero mode. If this integer value $V$ can be represented as a `long`, then the `result` is the `long value $V$`.
- Otherwise, either the `value'` must be too small (a negative value of large magnitude or negative infinity), and the `result` is the smallest representable value of type `long`, or the `value'` must be too large (a positive value of large magnitude or positive infinity), and the `result` is the largest representable value of type `long`.

**Notes**  
The `f2l` instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of `value'` and may also lose precision.
**fadd**

**Operation**
Add float

**Format**
```
fadd
```

**Forms**
`fadd = 98 (0x62)`

**Operand**
`..., value1, value2 ⇒`

**Stack**
`..., result`

**Description**
Both `value1` and `value2` must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in `value1'` and `value2'`. The float result is `value1' + value2'`. The result is pushed onto the operand stack.

The result of an `fadd` instruction is governed by the rules of IEEE arithmetic:

- If either `value1'` or `value2'` is NaN, the result is NaN.
- The sum of two infinities of opposite sign is NaN.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of an infinity and any finite value is equal to the infinity.
- The sum of two zeroes of opposite sign is positive zero.
- The sum of two zeroes of the same sign is the zero of that sign.
- The sum of a zero and a nonzero finite value is equal to the nonzero value.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
In the remaining cases, where neither operand is an infinity, a zero, or NaN and the values have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a `float`, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a `float`, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an `fadd` instruction never throws a runtime exception.
**faload**

**Operation**  Load float from array

**Format**

```
faload
```

**Forms**

`faload = 48 (0x30)`

**Operand Stack**

`..., arrayref, index ⇒...
value`

**Description**

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

**Runtime Exceptions**

If `arrayref` is `null`, `faload` throws a `NullPointerException`.

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

**Operation**  
Store into float array

**Format**  

**Forms**  
\[
\text{fastore} = 81 \ (0x51)
\]

**Operand Stack**  
\[
\ldots, \text{arrayref}, \text{index}, \text{value} \Rightarrow 
\]

**Description**  
The arrayref must be of type reference and must refer to an array whose components are of type float. The index must be of type int, and the value must be of type float. The arrayref, index, and value are popped from the operand stack. The float value undergoes value set conversion (§3.8.3), resulting in value', and value' is stored as the component of the array indexed by index.

**Runtime Exceptions**  
If arrayref is null, fastore throws a NullPointerException. Otherwise, if index is not within the bounds of the array referenced by arrayref, the fastore instruction throws an ArrayIndexOutOfBoundsException.
**fcmp<op>**        **fcmp<op>**

**Operation**  Compare float

**Format**  

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compare float</th>
</tr>
</thead>
</table>

**Forms**  

- `fcmpg = 150 (0x96)`
- `fcmpl = 149 (0x95)`

**Operand**  

- `...`<br>`value1, value2`<br>`⇒`

**Stack**  

- `...`<br>`result`

**Description**  Both `value1` and `value2` must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in `value1'` and `value2'`. A floating-point comparison is performed:

  - If `value1'` is greater than `value2'`, the int value `1` is pushed onto the operand stack.
  - Otherwise, if `value1'` is equal to `value2'`, the int value `0` is pushed onto the operand stack.
  - Otherwise, if `value1'` is less than `value2'`, the int value `-1` is pushed onto the operand stack.
  - Otherwise, at least one of `value1'` or `value2'` is NaN. The `fcmpg` instruction pushes the int value `1` onto the operand stack and the `fcmpl` instruction pushes the int value `-1` onto the operand stack.

Floating-point comparison is performed in accordance with IEEE 754. All values other than NaN are ordered, with negative infinity less than all finite values and positive infinity greater than all finite values. Positive zero and negative zero are considered equal.
Notes

The `fcmpg` and `fcmpl` instructions differ only in their treatment of a comparison involving NaN. NaN is unordered, so any float comparison fails if either or both of its operands are NaN. With both `fcmpg` and `fcmpl` available, any float comparison may be compiled to push the same result onto the operand stack whether the comparison fails on non-NaN values or fails because it encountered a NaN. For more information, see §7.5, “More Control Examples.”
\( f\text{const}_<f> \) \hspace{1cm} \( f\text{const}_<f> \)

**Operation**  
Push float

**Format**  
\( f\text{const}_<f> \)

**Forms**  
\( f\text{const}_0 = 11 \text{ (0xb)} \)
\( f\text{const}_1 = 12 \text{ (0xc)} \)
\( f\text{const}_2 = 13 \text{ (0xd)} \)

**Operand**  
\( \ldots \Rightarrow \)

**Stack**  
\( \ldots, <f> \)

**Description**  
Push the float constant \(<f>\) (0.0, 1.0, or 2.0) onto the operand stack.
### fdiv

**Operation**
Divide float

**Format**
```
| fdiv |
```

**Forms**
\[ fdiv = 110 \ (0x6e) \]

**Operand Stack**
\[ ..., value1, value2 \Rightarrow \]

**Stack**
\[ ..., result \]

**Description**
Both `value1` and `value2` must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in `value1'` and `value2'`. The float result is `value1' / value2'`. The result is pushed onto the operand stack.

The result of an `fdiv` instruction is governed by the rules of IEEE arithmetic:

- If either `value1'` or `value2'` is NaN, the result is NaN.

- If neither `value1'` nor `value2'` is NaN, the sign of the result is positive if both values have the same sign, negative if the values have different signs.

- Division of an infinity by an infinity results in NaN.

- Division of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.

- Division of a finite value by an infinity results in a signed zero, with the sign-producing rule just given.

- Division of a zero by a zero results in NaN; division of zero by any other finite value results in a signed zero, with the sign-producing rule just given.

- Division of a nonzero finite value by a zero results in a signed infinity, with the sign-producing rule just given.
fdiv (cont.)  

- In the remaining cases, where neither operand is an infinity, a zero, or NaN, the quotient is computed and rounded to the nearest float using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a float, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a float, we say the operation underflows; the result is then a zero of appropriate sign.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, division by zero, or loss of precision may occur, execution of an fdiv instruction never throws a runtime exception.
### fload

**Operation**
Load float from local variable

**Format**

<table>
<thead>
<tr>
<th>fload</th>
<th>index</th>
</tr>
</thead>
</table>

**Forms**

\[
fload = 23 \ (0x17)
\]

**Operand**
... ⇒

**Stack**
... , value

**Description**
The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The local variable at index must contain a float. The value of the local variable at index is pushed onto the operand stack.

**Notes**
The fload opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
\textit{fload\_<n>} \hspace{1cm} \textit{fload\_<n>}

\begin{itemize}
  \item \textbf{Operation}: Load float from local variable
  \item \textbf{Format}: \texttt{fload\_<n>}
  \item \textbf{Forms}:
    \begin{align*}
    \text{fload}_0 &= 34 \ (0x22) \\
    \text{fload}_1 &= 35 \ (0x23) \\
    \text{fload}_2 &= 36 \ (0x24) \\
    \text{fload}_3 &= 37 \ (0x25)
    \end{align*}
  \item \textbf{Operand Stack}: \ldots \Rightarrow \ldots, \textit{value}
  \item \textbf{Description}: The \textit{<n>} must be an index into the local variable array of the current frame (§3.6). The local variable at \textit{<n>} must contain a float. The \textit{value} of the local variable at \textit{<n>} is pushed onto the operand stack.
  \item \textbf{Notes}: Each of the \textit{fload\_<n>} instructions is the same as \textit{fload} with an \textit{index} of \textit{<n>}, except that the operand \textit{<n>} is implicit.
\end{itemize}
**fmul**

**Operation** Multiply float

**Format**

```
fmul
```

**Forms**

\[ fmul = 106 \ (0x6a) \]

**Operand Stack**

\[ \ldots, \text{value1}, \text{value2} \Rightarrow \ldots, \text{result} \]

**Description** Both \textit{value1} and \textit{value2} must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \textit{value1}' and \textit{value2}'. The float result is \textit{value1}' \times \textit{value2}'. The result is pushed onto the operand stack.

The result of an \textit{fmul} instruction is governed by the rules of IEEE arithmetic:

- If either \textit{value1}' or \textit{value2}' is NaN, the result is NaN.
- If neither \textit{value1}' nor \textit{value2}' is NaN, the sign of the result is positive if both values have the same sign, and negative if the values have different signs.
- Multiplication of an infinity by a zero results in NaN.
- Multiplication of an infinity by a finite value results in a signed infinity, with the sign-producing rule just given.
- In the remaining cases, where neither an infinity nor NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round to nearest mode. If the magnitude is too large to represent as a float, we say the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent as a float, we say the operation underflows; the result is then a zero of appropriate sign.
The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an $fmul$ instruction never throws a runtime exception.
fneg

**Operation**  
Negate float

**Format**  

```
fneg
```

**Forms**  

\[ fneg = 118 \ (0x76) \]

**Operand Stack**  

\[ \ldots, value \Rightarrow \ldots \]

**Description**  
The value must be of type float. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in value'. The float result is the arithmetic negation of value'. This result is pushed onto the operand stack.

For float values, negation is not the same as subtraction from zero. If \( x \) is +0.0, then \( 0.0 - x \) equals +0.0, but \( -x \) equals −0.0. Unary minus merely inverts the sign of a float.

Special cases of interest:

- If the operand is NaN, the result is NaN (recall that NaN has no sign).
- If the operand is an infinity, the result is the infinity of opposite sign.
- If the operand is a zero, the result is the zero of opposite sign.
**frem**

**Operation**  
Remainder float

**Format**

```
frem
```

**Forms**

\( \textit{frem} = 114 \) \((0x72)\)

**Operand**

\( ..., \text{value1, value2} \Rightarrow \)

**Stack**

\( ..., \text{result} \)

**Description**  
Both \textit{value1} and \textit{value2} must be of type \texttt{float}. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in \textit{value1}' and \textit{value2}'. The \textit{result} is calculated and pushed onto the operand stack as a \texttt{float}.

The \textit{result} of an \textit{frem} instruction is not the same as that of the so-called remainder operation defined by IEEE 754. The IEEE 754 “remainder” operation computes the remainder from a rounding division, not a truncating division, and so its behavior is \textit{not} analogous to that of the usual integer remainder operator. Instead, the Java virtual machine defines \textit{frem} to behave in a manner analogous to that of the Java virtual machine integer remainder instructions (\textit{irem} and \textit{lrem}); this may be compared with the C library function \texttt{fmod}.

The result of an \textit{frem} instruction is governed by these rules:

- If either \textit{value1}' or \textit{value2}' is NaN, the result is NaN.
- If neither \textit{value1}' nor \textit{value2}' is NaN, the sign of the result equals the sign of the dividend.
- If the dividend is an infinity or the divisor is a zero or both, the result is NaN.
- If the dividend is finite and the divisor is an infinity, the result equals the dividend.
If the dividend is a zero and the divisor is finite, the result equals the dividend.

In the remaining cases, where neither operand is an infinity, a zero, or NaN, the floating-point remainder result from a dividend \( \textit{value1}' \) and a divisor \( \textit{value2}' \) is defined by the mathematical relation \( \text{result} = \textit{value1}' - (\textit{value2}' \times q) \), where \( q \) is an integer that is negative only if \( \textit{value1}' / \textit{value2}' \) is negative and positive only if \( \textit{value1}' / \textit{value2}' \) is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of \( \textit{value1}' \) and \( \textit{value2}' \).

Despite the fact that division by zero may occur, evaluation of an \textit{frem} instruction never throws a runtime exception. Overflow, underflow, or loss of precision cannot occur.

**Notes**

The IEEE 754 remainder operation may be computed by the library routine \texttt{Math.IEEEremainder}. 

\textit{frem} (cont.)
The current method must have return type `float`. The `value` must be of type `float`. If the current method is a *synchronized* method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a `monitorexit` instruction in the current thread. If no exception is thrown, `value` is popped from the operand stack of the current frame (§3.6) and undergoes value set conversion (§3.8.3), resulting in `value'`. The `value'` is 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 interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

If the virtual machine implementation does not enforce the rules on structured locking described in §3.11.10, then if the current method is a *synchronized* method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, `freturn` throws an `IllegalMonitorStateException`. This can happen, for example, if a synchronized method contains a `monitorexit` instruction, but no `monitorenter` instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the first of those rules is violated during invocation of the current method, then `freturn` throws an `IllegalMonitorStateException`.
\textbf{fstore} \hspace{2cm} fstore

\textbf{Operation} \hspace{1cm} Store \texttt{float} into local variable

\textbf{Format} \hspace{2cm}
\begin{tabular}{|l|}
\hline
\textit{fstore} \\
\hline
\textit{index} \\
\hline
\end{tabular}

\textbf{Forms} \hspace{1cm} \textit{fstore} = 56 (0x38)

\textbf{Operand} \hspace{1cm} \ldots, \textit{value} \Rightarrow

\textbf{Stack} \hspace{1cm} \ldots

\textbf{Description} \hspace{1cm} The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The value on the top of the operand stack must be of type \texttt{float}. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in \textit{value′}. The value of the local variable at index is set to \textit{value′}.

\textbf{Notes} \hspace{1cm} The \textit{fstore} opcode can be used in conjunction with the \textit{wide} instruction to access a local variable using a two-byte unsigned index.
\textbf{fstore}_<n> \hspace{1cm} \textbf{fstore}_<n>

**Operation**  
Store float into local variable

**Format**
\[ \text{fstore}_<n> \]

**Forms**
- \text{fstore}_0 = 67 (0x43)
- \text{fstore}_1 = 68 (0x44)
- \text{fstore}_2 = 69 (0x45)
- \text{fstore}_3 = 70 (0x46)

**Operand Stack**
\[ \ldots, \text{value} \Rightarrow \]

**Description**  
The \(<n>\) must be an index into the local variable array of the current frame (§3.6). The \textit{value} on the top of the operand stack must be of type float. It is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in \textit{value}'\textquoteleft. The value of the local variable at \(<n>\) is set to \textit{value}'\textquoteleft.

**Notes**  
Each of the \textit{fstore}_<n> is the same as \textit{fstore} with an index of \(<n>\), except that the operand \(<n>\) is implicit.
\textbf{fsub} \hspace{1cm} \textbf{fsub}

**Operation** Subtract float

**Format** \[fsub\]

**Forms** \[fsub = 102 \ (0x66)\]

**Operand** \[, value1, value2 \Rightarrow\]

**Stack** \[, result\]

**Description** Both value1 and value2 must be of type float. The values are popped from the operand stack and undergo value set conversion (§3.8.3), resulting in value1’ and value2’. The float result is value1’ − value2’. The result is pushed onto the operand stack.

For float subtraction, it is always the case that \(a - b\) produces the same result as \(a + (-b)\). However, for the fsub instruction, subtraction from zero is not the same as negation, because if \(x\) is +0.0, then 0.0−x equals +0.0, but −x equals −0.0.

The Java virtual machine requires support of gradual underflow as defined by IEEE 754. Despite the fact that overflow, underflow, or loss of precision may occur, execution of an fsub instruction never throws a runtime exception.
### `getfield`

**Operation**
Fetch field from object

**Format**

<table>
<thead>
<tr>
<th>getfield</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**

\[\text{getfield} = 180 \ (0xb4)\]

**Operand**

\[\ldots, \text{objectref} \Rightarrow\]

**Stack**

\[\ldots, \text{value}\]

**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 runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) \mid indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The referenced field is resolved (§5.4.3.3). The `value` of the referenced field in `objectref` is fetched and pushed onto the operand stack.

The type of `objectref` must not be an array type. If the field is `protected` (§4.6), and it is a member of a superclass of the current class, and the field is not declared in the same runtime package (§5.3) as the current class, then the class of `objectref` must be either the current class or a subclass of the current class.

**Linking**

**Exceptions**

During resolution of the symbolic reference to the field, any of the errors pertaining to field resolution documented in §5.4.3.3 can be thrown.

Otherwise, if the resolved field is a static field, `getfield` throws an `IncompatibleClassChangeError`.

---

**Note:**
This page contains an excerpt from the The Java Virtual Machine Instruction Set manual. The content is for educational purposes and is not intended for commercial use without written permission from Sun Microsystems, Inc.
**getfield (cont.)**

**Runtime** Otherwise, if `objectref` is `null`, the `getfield` instruction throws a `NullPointerException`.

**Notes** The `getfield` instruction cannot be used to access the `length` field of an array. The `arraylength` instruction is used instead.
**getstatic**

**Operation**
Get static field from class

**Format**

<table>
<thead>
<tr>
<th>getstatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**
getstatic = 178 (0xb2)

**Operand Stack**

..., ⇒
value

**Description**
The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.3).

On successful resolution of the field, the class or interface that declared the resolved field is initialized (§5.5) if that class or interface has not already been initialized.

The value of the class or interface field is fetched and pushed onto the operand stack.

**Linking Exceptions**
During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution documented in Section 5.4.3.3 can be thrown.

Otherwise, if the resolved field is not a static (class) field or an interface field, `getstatic` throws an IncompatibleClassChangeError.

**Runtime Exception**
Otherwise, if execution of this `getstatic` instruction causes initialization of the referenced class or interface, `getstatic` may throw an Error as detailed in Section 5.5.
**goto**

**Operation**  Branch always

**Format**

<table>
<thead>
<tr>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

**Forms**  goto = 167 (0xa7)

**Operand Stack**  No change

**Description**  The unsigned bytes `branchbyte1` and `branchbyte2` are used to construct a signed 16-bit `branchoffset`, where `branchoffset` is `(branchbyte1 << 8) | branchbyte2`. Execution proceeds at that offset from the address of the opcode of this `goto` instruction. The target address must be that of an opcode of an instruction within the method that contains this `goto` instruction.
**go**

**Operation**
Branch always (wide index)

**Format**

<table>
<thead>
<tr>
<th>Branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto_w</td>
</tr>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
<tr>
<td>branchbyte3</td>
</tr>
<tr>
<td>branchbyte4</td>
</tr>
</tbody>
</table>

**Forms**
goto_w = 200 (0xc8)

**Operand**
No change

**Stack**

**Description**
The unsigned bytes `branchbyte1`, `branchbyte2`, `branchbyte3`, and `branchbyte4` are used to construct a signed 32-bit `branchoffset`, where `branchoffset` is `(branchbyte1 << 24) | (branchbyte2 << 16) | (branchbyte3 << 8) | branchbyte4`. Execution proceeds at that offset from the address of the opcode of this `goto_w` instruction. The target address must be that of an opcode of an instruction within the method that contains this `goto_w` instruction.

**Notes**
Although the `goto_w` instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes (§4.11). This limit may be raised in a future release of the Java virtual machine.
### i2b

**Operation** Convert `int` to `byte`

**Format**

```
i2b
```

**Forms**

\[ i2b = 145 \ (0x91) \]

**Operand** \[ \ldots, \text{value} \Rightarrow \]

**Stack** \[ \ldots, \text{result} \]

**Description** The `value` on the top of the operand stack must be of type `int`. It is popped from the operand stack, truncated to a `byte`, then sign-extended to an `int` `result`. That `result` is pushed onto the operand stack.

**Notes** The `i2b` instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of `value`. The `result` may also not have the same sign as `value`. 
### i2c

**Operation**  Convert int to char

**Format**

```
i2c
```

**Forms**  

\[ i2c = 146 \ (0x92) \]

**Operand**  

\[ \ldots, \text{value} \Rightarrow \]

**Stack**  

\[ \ldots, \text{result} \]

**Description**  
The \textit{value} on the top of the operand stack must be of type \textit{int}. It is popped from the operand stack, truncated to \textit{char}, then zero-extended to an \textit{int} \textit{result}. That \textit{result} is pushed onto the operand stack.

**Notes**  
The \textit{i2c} instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of \textit{value}. The \textit{result} (which is always positive) may also not have the same sign as \textit{value}. 
\textbf{Operation} \hspace{1cm} \textbf{Convert int to double} \\

\textbf{Format} \hspace{1cm} \textit{i2d} \\

\textbf{Forms} \hspace{1cm} i2d = 135 (0x87) \\

\textbf{Operand} \hspace{1cm} \ldots, value \Rightarrow \\

\textbf{Stack} \hspace{1cm} \ldots, result \\

\textbf{Description} \hspace{1cm} The value on the top of the operand stack must be of type int. It is popped from the operand stack and converted to a double result. The result is pushed onto the operand stack.

\textbf{Notes} \hspace{1cm} The \textit{i2d} instruction performs a widening primitive conversion (JLS3 §5.1.2). Because all values of type int are exactly representable by type double, the conversion is exact.
**Operation**  
Convert int to float

**Format**

```
i2f
```

**Forms**

$i2f = 134$ (0x86)

**Operand**

\[ \ldots, \text{value} \Rightarrow \ldots \]

**Stack**

\[ \ldots, \text{result} \]

**Description**

The value on the top of the operand stack must be of type int. It is popped from the operand stack and converted to the float result using IEEE 754 round to nearest mode. The result is pushed onto the operand stack.

**Notes**

The $i2f$ instruction performs a widening primitive conversion (JLS3 §5.1.2), but may result in a loss of precision because values of type float have only 24 significand bits.
**i2l**

**Operation**  Convert int to long

**Format**

\[
\text{\texttt{i2l}}
\]

**Forms**  \(i2l = 133\) (0x85)

**Operand**  \(\ldots, value \Rightarrow\)

**Stack**  \(\ldots, result\)

**Description**  The value on the top of the operand stack must be of type int. It is popped from the operand stack and sign-extended to a long result. That result is pushed onto the operand stack.

**Notes**  The \(i2l\) instruction performs a widening primitive conversion (JLS3 §5.1.2). Because all values of type int are exactly representable by type long, the conversion is exact.
### i2s

**Operation**  Convert int to short

**Format**

```
   i2s
```

**Forms**

\( i2s = 147 \ (0x93) \)

**Operand**

\( \ldots, \text{value} \Rightarrow \)

**Stack**

\( \ldots, \text{result} \)

**Description**

The value on the top of the operand stack must be of type int. It is popped from the operand stack, truncated to a short, then sign-extended to an int result. That result is pushed onto the operand stack.

**Notes**

The \textit{i2s} instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of value. The result may also not have the same sign as value.
**iadd**

**Operation**
Add int

**Format**
iadd

**Forms**
iadd = 96 (0x60)

**Operand**
..., value1, value2 ⇒

**Stack**
..., result

**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.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type int. 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.

Despite the fact that overflow may occur, execution of an iadd instruction never throws a runtime exception.
**iaload**

**Operation**  Load int from array

**Format**  

\[
\text{iaload}
\]

**Forms**  \( iaload = 46 \ (0x2e) \)

**Operand Stack**  \( \ldots, \text{arrayref}, \text{index} \Rightarrow \ldots, \text{value} \)

**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 int. 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 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.
**iand**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Boolean AND int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td><code>iand</code></td>
</tr>
<tr>
<td>Forms</td>
<td><code>iand = 126 (0x7e)</code></td>
</tr>
<tr>
<td>Operand Stack</td>
<td><code>..., value1, value2 ⇒</code></td>
</tr>
<tr>
<td>Stack</td>
<td><code>..., result</code></td>
</tr>
</tbody>
</table>

**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.
iastore

Operation  Store into int array

Format  

Forms  iastore = 79 (0x4f)

Operand  ..., arrayref, index, value ⇒

Stack  ...

Description  The arrayref must be of type reference and must refer to an array whose components are of type int. Both index and value must be of type int. The arrayref, index, and value are popped from the operand stack. The int value is stored as the component of the array indexed by index.

Runtime Exceptions  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.
**iconst_<i>**

**Operation**  Push int constant

**Format**  

<table>
<thead>
<tr>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>iconst_&lt;i&gt;</td>
</tr>
</tbody>
</table>

**Forms**  

- iconst_m1 = 2 (0x2)
- iconst_0 = 3 (0x3)
- iconst_1 = 4 (0x4)
- iconst_2 = 5 (0x5)
- iconst_3 = 6 (0x6)
- iconst_4 = 7 (0x7)
- iconst_5 = 8 (0x8)

**Operand**  

... ⇒

**Stack**  

..., <i>

**Description**  Push the int constant <i> (-1, 0, 1, 2, 3, 4 or 5) onto the operand stack.

**Notes**  Each of this family of instructions is equivalent to bipush <i> for the respective value of <i>, except that the operand <i> is implicit.
**idiv**

**Operation**  Divide int

**Format**

```markdown
idiv
```

**Forms**

```
idiv = 108 (0x6c)
```

**Operand**  ...
**Stack**  ...

**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 programming language expression value1 / value2. The result is pushed onto the operand stack.

An int division rounds towards 0; that is, the quotient produced for int values in \( n/d \) is an int value \( q \) whose magnitude is as large as possible while satisfying \( |d \cdot q| \leq |n| \). Moreover, \( q \) is positive when \( |n| \geq |d| \) and \( n \) and \( d \) have the same sign, but \( q \) is negative when \( |n| \geq |d| \) 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 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.
```plaintext
if_acmp<cond>                     if_acmp<cond>

Operation  Branch if reference comparison succeeds

Format

<table>
<thead>
<tr>
<th>if_acmp&lt;cond&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

Forms

if_acmpeq = 165 (0xa5)
if_acmpne = 166 (0xa6)

Operand

..., value1, value2 ⇒

Stack

...

Description

Both value1 and value2 must be of type reference. They are both popped from the operand stack and compared. The results of the comparison 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 branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be (branchbyte1 << 8) | branchbyte2. Execution then 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, if the comparison fails, execution proceeds at the address of the instruction following this if_acmp<cond> instruction.
```
if_icmp<cond>

**Operation**  
Branch if int comparison succeeds

**Format**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Branch if int comparison succeeds</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if_icmp&lt;cond&gt;</strong></td>
<td>branchbyte1</td>
</tr>
<tr>
<td></td>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

**Forms**

- `if_icmpeq` = 159 (0x9f)
- `if_icmpne` = 160 (0xa0)
- `if_icmplt` = 161 (0xa1)
- `if_icmpge` = 162 (0xa2)
- `if_icmpgt` = 163 (0xa3)
- `if_icmple` = 164 (0xa4)

**Operand**

... , value1, value2 ⇒

**Stack**

... 

**Description**

Both value1 and value2 must be of type int. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparison 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_icmp<cond> (cont.)

If the comparison succeeds, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be (branchbyte1 << 8) | branchbyte2. Execution then proceeds at that offset from the address of the opcode of this if_icmp<cond> instruction. The target address must be that of an opcode of an instruction within the method that contains this if_icmp<cond> instruction.

Otherwise, execution proceeds at the address of the instruction following this if_icmp<cond> instruction.
if<cond>

Operation Branch if int comparison with zero succeeds

Format

<table>
<thead>
<tr>
<th>if&lt;cond&gt;</th>
<th>branchbyte1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

Forms

<table>
<thead>
<tr>
<th>Forms</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifeq</td>
<td>153 (0x99)</td>
</tr>
<tr>
<td>ifne</td>
<td>154 (0x9a)</td>
</tr>
<tr>
<td>iflt</td>
<td>155 (0x9b)</td>
</tr>
<tr>
<td>ifge</td>
<td>156 (0x9c)</td>
</tr>
<tr>
<td>ifgt</td>
<td>157 (0x9d)</td>
</tr>
<tr>
<td>ifle</td>
<td>158 (0x9e)</td>
</tr>
</tbody>
</table>

Operand ...

Stack ...

Description The value must be of type int. 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<cond> (cont.) if<cond> (cont.)

If the comparison succeeds, the unsigned `branchbyte1` and `branchbyte2` are used to construct a signed 16-bit offset, where the offset is calculated to be `(branchbyte1 << 8) | branchbyte2`. Execution then 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.
**ifnonnull**

**Operation**
Branch if reference not null

**Format**
```
+---+=
| ifnonnull |
| branchbyte1 |
| branchbyte2 |
```

**Forms**
`ifnonnull = 199 (0xc7)`

**Operand**
... , value ⇒

**Stack**
...

**Description**
The value must be of type reference. It is popped from the operand stack. If value is not null, the unsigned `branchbyte1` and `branchbyte2` are used to construct a signed 16-bit offset, where the offset is calculated to be `(branchbyte1 << 8) | branchbyte2`. Execution then 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.
**ifnull**

**Operation**  
Branch if reference is null

**Format**

<table>
<thead>
<tr>
<th>ifnull</th>
</tr>
</thead>
<tbody>
<tr>
<td>branchbyte1</td>
</tr>
<tr>
<td>branchbyte2</td>
</tr>
</tbody>
</table>

**Forms**  
ifnull = 198 (0xc6)

**Operand**  
..., value ⇒

**Stack**  
...

**Description**  
The value must of type reference. It is popped from the operand stack. If value is null, the unsigned branchbyte1 and branchbyte2 are used to construct a signed 16-bit offset, where the offset is calculated to be \((\text{branchbyte1} \ll 8) \| \text{branchbyte2}\). Execution then 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.
**iinc**

**Operation**  Increment local variable by constant

**Format**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>iinc</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>const</td>
</tr>
</tbody>
</table>

**Forms**  \( iinc = 132 \) (0x84)

**Operand Stack**  No change

**Description**  The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The const is an immediate signed byte. The local variable at index must contain an int. The value const is first sign-extended to an int, and then the local variable at index is incremented by that amount.

**Notes**  The iinc opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index and to increment it by a two-byte immediate value.
\textbf{iload} \hspace{1cm} \textbf{iload}

\begin{description}
\item[Operation] Load int from local variable
\item[Format] 
\begin{tabular}{|c|}
\hline
\textit{iload} \\
\hline
\textit{index} \\
\hline
\end{tabular}
\item[Forms] \textit{iload} = 21 (0x15)
\item[Operand] \ldots \Rightarrow
\item[Stack] \ldots, value
\item[Description] The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The local variable at index must contain an int. The value of the local variable at index is pushed onto the operand stack.
\item[Notes] The iload opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
\end{description}
### iload_<n>

**Operation**  
Load int from local variable

**Format**

```
  iload_<n>
```

**Forms**

- `iload_0 = 26 (0x1a)`
- `iload_1 = 27 (0x1b)`
- `iload_2 = 28 (0x1c)`
- `iload_3 = 29 (0x1d)`

**Operand Stack**

```
... => ...
```

**Description**

The `<n>` must be an index into the local variable array of the current frame (§3.6). The local variable at `<n>` must contain an int. The value of the local variable at `<n>` 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.
**imul**

**Operation**  Multiply int

**Format**  

**Forms**  $imul = 104 \ (0x68)$

**Operand Stack**  

... , value1, value2 ⇒

**Stack**  

... , result

**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.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type int. 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.

Despite the fact that overflow may occur, execution of an imul instruction never throws a runtime exception.
**ineg**

**Operation**  Negate int

**Format**  

```
 ine
```

**Forms**  

\[ ine = 116 \ (0x74) \]

**Operand**  

\[ ..., value \Rightarrow \]

**Stack**  

\[ ..., result \]

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

For \textit{int} values, negation is the same as subtraction from zero. Because the Java 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 \textit{int} results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all \textit{int} values \(x\), \(-x\) equals \((\sim x) + 1\).
The `instanceof` instruction determines if an object is of a given type. It is used to check if the object is an instance of the resolved class, array, or interface type.

**Operation**
Determine if object is of given type

**Format**

<table>
<thead>
<tr>
<th><code>instanceof</code></th>
<th><code>indexbyte1</code></th>
<th><code>indexbyte2</code></th>
</tr>
</thead>
</table>

**Forms**
`instanceof = 193 (0xc1)`

**Operand**
`. .., objectref ⇒`

**Stack**
`. .., result`

**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 runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type.

If `objectref` is `null`, the `instanceof` instruction pushes an `int result` of 0 as an int on the operand stack.

Otherwise, the named class, array, or interface type is resolved (§5.4.3.1). If `objectref` is an instance of the resolved class or array or implements the resolved interface, the `instanceof` instruction pushes an `int result` of 1 as an int on the operand stack; otherwise, it pushes an `int 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 an ordinary (nonarray) class, then:
  - If `T` is a class type, then `S` must be the same class as `T` or a subclass of `T`.
  - If `T` is an interface type, then `S` must implement interface `T`. 
**instanceof (cont.)**

- If \( S \) is an interface type, then:
  - If \( T \) is a class type, then \( T \) must be \texttt{Object}.
  - If \( T \) is an interface type, then \( T \) must be the same interface as \( S \), or a superinterface of \( S \).

- If \( S \) is a class representing the array type \( SC[] \), that is, an array of components of type \( SC \), then:
  - If \( T \) is a class type, then \( T \) must be \texttt{Object}.
  - If \( T \) is an array type \( TC[] \), that is, an array of components of type \( TC \), then one of the following must be true:
    - \( TC \) and \( SC \) are the same primitive type.
    - \( TC \) and \( SC \) are reference types, and type \( SC \) can be cast to \( TC \) by these runtime rules.
  - If \( T \) is an interface type, \( T \) must be one of the interfaces implemented by arrays (JLS3 §4.10.3).

**Linking Exceptions**

During resolution of symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

**Notes**

The \texttt{instanceof} instruction is very similar to the \texttt{checkcast} instruction. It differs in its treatment of \texttt{null}, its behavior when its test fails (\texttt{checkcast} throws an exception, \texttt{instanceof} pushes a result code), and its effect on the operand stack.
**invokeinterface**  

**Operation**  
Invoke interface method

**Format**

```
  invokeinterface
  indexbyte1
  indexbyte2
  count
  0
```

**Forms**  
```
invokeinterface = 185 (0xb9)
```

**Operand**  
```
..., objectref, [arg1, [arg2 ...]] ⇒
```

**Stack**  
```
...
```

**Description**  
The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to an interface method (§5.1), which gives the name and descriptor (§4.3.3) of the interface method as well as a symbolic reference to the interface in which the interface method is to be found. The named interface method is resolved (§5.4.3.5). The interface method must not be an instance initialization method (§3.9) or the class or interface initialization method (§3.9).

The `count` operand is an unsigned byte that must not be zero. The `objectref` must be of type reference and must be followed on the operand stack by `nargs` argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved interface method. The value of the fourth operand byte must always be zero.

Let \( C \) be the class of `objectref`. The actual method to be invoked is selected by the following lookup procedure:
invokeinterface (cont.) invokeinterface (cont.)

- If \( C \) contains a declaration for an instance method with the same name and descriptor as the resolved method, then this is the method to be invoked, and the lookup procedure terminates.

- Otherwise, if \( C \) has a superclass, this same lookup procedure is performed recursively using the direct superclass of \( C \); the method to be invoked is the result of the recursive invocation of this lookup procedure.

- Otherwise, an AbstractMethodError is raised.

If the method is synchronized, the monitor associated with \( \text{objectref} \) is entered or reentered as if by execution of a monitorenter instruction in the current thread.

If the method is not native, the \( \text{nargs} \) argument values and \( \text{objectref} \) are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The \( \text{objectref} \) and the argument values are consecutively made the values of local variables of the new frame, with \( \text{objectref} \) in local variable 0, \( \text{arg1} \) in local variable 1 (or, if \( \text{arg1} \) is of type \text{long} or \text{double}, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine \( \text{pc} \) is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The \( \text{nargs} \) argument values and \( \text{objectref} \) are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns:
\textit{invokeinterface} (cont.) \hspace{1cm} \textit{invokeinterface} (cont.)

- If the native method is synchronized, the monitor associated with \texttt{objectref} is updated and possibly exited as if by execution of a \texttt{monitor-exit} instruction in the current thread.

- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

\textbf{Linking Exceptions}

During resolution of the symbolic reference to the interface method, any of the exceptions documented in §5.4.3.5 can be thrown.

\textbf{Runtime Exceptions}

Otherwise, if \texttt{objectref} is null, the \textit{invokeinterface} instruction throws a \texttt{NullPointerException}.

Otherwise, if the class of \texttt{objectref} does not implement the resolved interface, \textit{invokeinterface} throws an \texttt{IncompatibleClassChangeError}.

Otherwise, if no method matching the resolved name and descriptor is selected, \textit{invokeinterface} throws an \texttt{AbstractMethodError}.

Otherwise, if the selected method is not public, \textit{invokeinterface} throws an \texttt{IllegalAccessException}.

Otherwise, if the selected method is abstract, \textit{invokeinterface} throws an \texttt{AbstractMethodError}.

Otherwise, if the selected method is native and the code that implements the method cannot be bound, \textit{invokeinterface} throws an \texttt{UnsatisfiedLinkError}.
invokeinterface (cont.)

Notes

The count operand of the invokeinterface instruction records a measure of the number of argument values, where an argument value of type long or type double contributes two units to the count value and an argument of any other type contributes one unit. This information can also be derived from the descriptor of the selected method. The redundancy is historical.

The fourth operand byte exists to reserve space for an additional operand used in certain of Sun’s implementations, which replace the invokeinterface instruction by a specialized pseudo-instruction at runtime. It must be retained for backwards compatibility.

The nargs argument values and objectref are not one-to-one with the first nargs + 1 local variables. Argument values of types long and double must be stored in two consecutive local variables, thus more than nargs local variables may be required to pass nargs argument values to the invoked method.
**invokespecial**

**Operation**
Invoke instance method; special handling for superclass, private, and instance initialization method invocations

**Format**
```
invokespecial
indexbyte1
indexbyte2
```

**Forms**
`invokespecial = 183 (0xb7)`

**Operand**
`..., objectref, [arg1, [arg2 ...]] ⇒ ...

**Stack**
`...`

**Description**
The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.4). Finally, if the resolved method is protected (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same runtime package (§5.3) as the current class, then the class of `objectref` must be either the current class or a subclass of the current class.

Next, the resolved method is selected for invocation unless all of the following conditions are true:

- The ACC_SUPER flag (see Table 4.1, “Class access and property modifiers”) is set for the current class.
- The class of the resolved method is a superclass of the current class.
- The resolved method is not an instance initialization method (§3.9).
If the above conditions are true, the actual method to be invoked is selected by the following lookup procedure. Let $C$ be the direct superclass of the current class:

- If $C$ contains a declaration for an instance method with the same name and descriptor as the resolved method, then this method will be invoked. The lookup procedure terminates.

- Otherwise, if $C$ has a superclass, this same lookup procedure is performed recursively using the direct superclass of $C$. The method to be invoked is the result of the recursive invocation of this lookup procedure.

- Otherwise, an AbstractMethodError is raised.

The `objectref` must be of type `reference` and must be followed on the operand stack by `nargs` argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is synchronized, the monitor associated with `objectref` is entered or reentered as if by execution of a `monitorenter` instruction in the current thread.

If the method is not native, the `nargs` argument values and `objectref` are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The `objectref` and the argument values are consecutively made the values of local variables of the new frame, with `objectref` in local variable 0, `arg1` in local variable 1 (or, if `arg1` is of type `long` or `double`, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java 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.
**invokespecial (cont.)**

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The *nargs* argument values and *objectref* are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the native method is synchronized, the monitor associated with *objectref* is updated and possibly exited as if by execution of a *monitor-exit* instruction in the current thread.

- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

**Linking Exceptions**

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in §5.4.3.4 can be thrown.

Otherwise, if the resolved method is an instance initialization method, and the class in which it is declared is not the class symbolically referenced by the instruction, a *NoSuchMethodError* is thrown.

Otherwise, if the resolved method is a class (static) method, the *invokespecial* instruction throws an *IncompatibleClassChangeError*.

Otherwise, if no method matching the resolved name and descriptor is selected, *invokespecial* throws an *AbstractMethodError*.

Otherwise, if the selected method is abstract, *invokespecial* throws an *AbstractMethodError*. 
invokespecial (cont.)

Runtime Exceptions

Otherwise, if objectref is null, the invokespecial instruction throws a NullPointerException.

Otherwise, if the selected method is native and the code that implements the method cannot be bound, invokespecial throws an UnsatisfiedLinkError.

Notes

The difference between the invokespecial and the invokevirtual instructions is that invokevirtual invokes a method based on the class of the object. The invokespecial instruction is used to invoke instance initialization methods (§3.9) as well as private methods and methods of a superclass of the current class.

The invokespecial instruction was named invokenonvirtual prior to Sun’s JDK release 1.0.2.

The nargs argument values and objectref are not one-to-one with the first nargs + 1 local variables. Argument values of types long and double must be stored in two consecutive local variables, thus more than nargs local variables may be required to pass nargs argument values to the invoked method.
### invokestatic

**Operation**  
Invoke a class (static) method

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>invokestatic</td>
<td>indexbyte1</td>
</tr>
<tr>
<td></td>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**  
`invokestatic = 184 (0xb8)`

**Operand Stack**  
`..., [arg1, [arg2 ...]] ⇒`

**Description**  
The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.4). The method must not be an instance initialization method (§3.9) or the class or interface initialization method (§3.9). It must be static, and therefore cannot be abstract.

On successful resolution of the method, the class that declared the resolved method is initialized (§5.5) if that class has not already been initialized.

The operand stack must contain `nargs` argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved method.

If the method is synchronized, the monitor associated with the resolved `Class` object is entered or reentered as if by execution of a `monitorenter` instruction in the current thread.
invokestatic (cont.)

If the method is not native, the \textit{nargs} argument values are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The \textit{nargs} argument values are consecutively made the values of local variables of the new frame, with \textit{arg1} in local variable 0 (or, if \textit{arg1} is of type \texttt{long} or \texttt{double}, in local variables 0 and 1) and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine \texttt{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.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The \textit{nargs} argument values are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

- If the native method is \texttt{synchronized}, the monitor associated with the resolved \texttt{Class} object is updated and possibly exited as if by execution of a \texttt{monitorexit} instruction in the current thread.
- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

\textbf{Linking Exceptions}

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in §5.4.3.4 can be thrown.

Otherwise, if the resolved method is an instance method, the \texttt{invokestatic} instruction throws an \texttt{IncompatibleClassChangeError}.
invokestatic (cont.)

**Runtime Exceptions**

Otherwise, if execution of this `invokestatic` instruction causes initialization of the referenced class, `invokestatic` may throw an `Error` as detailed in §5.5.

Otherwise, if the resolved method is `native` and the code that implements the method cannot be bound, `invokestatic` throws an `UnsatisfiedLinkError`.

**Notes**

The `nargs` argument values are not one-to-one with the first `nargs` local variables. Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than `nargs` local variables may be required to pass `nargs` argument values to the invoked method.
**invokevirtual**  

**Operation**  
Invoke instance method; dispatch based on class

**Format**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>invokevirtual</td>
<td></td>
</tr>
<tr>
<td>indexbyte1</td>
<td></td>
</tr>
<tr>
<td>indexbyte2</td>
<td></td>
</tr>
</tbody>
</table>

**Forms**  
`invokevirtual = 182 (0xb6)`

**Operand**  
`..., objectref, [arg1, [arg2 ...]]` ⇒

**Stack**  
`...`

**Description**  
The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `

```
(indexbyte1 << 8) | indexbyte2
```

. The runtime constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.4). The method must not be an instance initialization method (§3.9) or the class or interface initialization method (§3.9). Finally, if the resolved method is protected (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same runtime package (§5.3) as the current class, then the class of `objectref` must be either the current class or a subclass of the current class.

Let `C` be the class of `objectref`. The actual method to be invoked is selected by the following lookup procedure:

- If `C` contains a declaration for an instance method `M` that overrides (§5.4.2.1) the resolved method, then `M` is the method to be invoked, and the lookup procedure terminates.
invokevirtual (cont.)

- Otherwise, if \( C \) has a superclass, this same lookup procedure is performed recursively using the direct superclass of \( C \); the method to be invoked is the result of the recursive invocation of this lookup procedure.

- Otherwise, an AbstractMethodError is raised.

The \texttt{objectref} must be followed on the operand stack by \texttt{nargs} argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is synchronized, the monitor associated with \texttt{objectref} is entered or reentered as if by execution of a \texttt{monitorenter} instruction in the current thread.

If the method is not native, the \texttt{nargs} argument values and \texttt{objectref} are popped from the operand stack. A new frame is created on the Java virtual machine stack for the method being invoked. The \texttt{objectref} and the argument values are consecutively made the values of local variables of the new frame, with \texttt{objectref} in local variable 0, \texttt{arg1} in local variable 1 (or, if \texttt{arg1} is of type \texttt{long} or \texttt{double}, in local variables 1 and 2), and so on. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java virtual machine \texttt{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.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java virtual machine, that is done. The \texttt{nargs} argument values and \texttt{objectref} are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§3.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:
**invokevirtual (cont.)**

- If the native method is synchronized, the monitor associated with `objectref` is updated and possibly exited as if by execution of a `monitor-exit` instruction in the current thread.

- If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

**Linked Exceptions**

During resolution of the symbolic reference to the method, any of the exceptions pertaining to method resolution documented in §5.4.3.4 can be thrown.

Otherwise, if the resolved method is a class (static) method, the `invokevirtual` instruction throws an `IncompatibleClassChangeError`.

**Runtime Exceptions**

Otherwise, if `objectref` is `null`, the `invokevirtual` instruction throws a `NullPointerException`.

Otherwise, if no method matching the resolved name and descriptor is selected, `invokevirtual` throws an `AbstractMethodError`. Otherwise, if the selected method is `abstract`, `invokevirtual` throws an `AbstractMethodError`.

Otherwise, if the selected method is `native` and the code that implements the method cannot be bound, `invokevirtual` throws an `UnsatisfiedLinkError`.

**Notes**

The `nargs` argument values and `objectref` are not one-to-one with the first `nargs + 1` local variables. Argument values of types `long` and `double` must be stored in two consecutive local variables, thus more than `nargs` local variables may be required to pass `nargs` argument values to the invoked method.
**ior**

**Operation**  Boolean OR int

**Format**  

| ior |

**Forms**  

\[ ior = 128 \ (0x80) \]

**Operand Stack**  

\[ \ldots, \text{value1}, \text{value2} \Rightarrow \]

\[ \ldots, \text{result} \]

**Description**  Both \text{value1} and \text{value2} must be of type int. They are popped from the operand stack. An int result is calculated by taking the bitwise inclusive OR of \text{value1} and \text{value2}. The result is pushed onto the operand stack.
**irem**

**Operation**  
Remainder int

**Format**

```
  irem
```

**Forms**

\[ irem = 112 \ (0x70) \]

**Operand Stack**

\[ \ldots, value1, value2 \Rightarrow \ldots, result \]

**Description**

Both \( value1 \) and \( value2 \) must be of type int. The values are popped from the operand stack. The int result is \( value1 \mod (value1 \div value2) \times value2 \). The result is pushed onto the operand stack.

The result of the \( irem \) instruction is such that \((a/b) \times b + (a \mod b)\) is equal to \( a \). This identity holds even in the special case in which 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 an int remainder operator is 0, \( irem \) throws an ArithmeticException.
**ireturn**

**Operation**  
Return `int` from method

**Format**  
```
ireturn
```

**Forms**  
`ireturn` = 172 (0xac)

**Operand Stack**  
`. . . , value ⇒`

**Description**  
The current method must have return type `boolean`, `byte`, `short`, `char`, or `int`. The `value` must be of type `int`. If the current method is a synchronized method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a `monitorexit` instruction in the current thread. If no exception is thrown, `value` is popped from the operand stack of the current frame (§3.6) 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 interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**  
If the virtual machine implementation does not enforce the rules on structured locking described in §3.11.10, then if the current method is a synchronized method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, `ireturn` throws an `IllegalMonitorStateException`. This can happen, for example, if a `synchronized` method contains a `monitorexit` instruction, but no `monitorenter` instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the first of those rules is violated during invocation of the current method, then `ireturn` throws an `IllegalMonitorStateException`. 
**ishl**

**Operation**  
Shift left int

**Format**  
```

ishl
```

**Forms**  
`ishl = 120 (0x78)`

**Operand Stack**  
`... , value1, value2 ⇒`

**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 5 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.
**ishr**

**Operation**  Arithmetic shift right \( \text{int} \)

**Format**  

```

| ishr |
```

**Forms**  \( ishr = 122 \ (0x7a) \)

**Operand**  \( ..., \text{value1}, \text{value2} \Rightarrow \)

**Stack**  \( ..., \text{result} \)

**Description**  Both \( \text{value1} \) and \( \text{value2} \) must be of type \( \text{int} \). The values are popped from the operand stack. An \( \text{int} \) result is calculated by shifting \( \text{value1} \) right by \( s \) bit positions, with sign extension, where \( s \) is the value of the low 5 bits of \( \text{value2} \). The result is pushed onto the operand stack.

**Notes**  The resulting value is \( \lfloor \frac{\text{value1}}{2^s} \rfloor \), where \( s \) is \( \text{value2} \& 0x1f \). For non-negative \( \text{value1} \), this is equivalent to truncating \( \text{int} \) division by 2 to the power \( s \). The shift distance actually used is always in the range 0 to 31, inclusive, as if \( \text{value2} \) were subjected to a bitwise logical AND with the mask value 0x1f.
**istore**

**Operation**  Store int into local variable

**Format**

```
               istore
               index
```

**Forms**  `istore = 54 (0x36)`

**Operand**  `…, value ⇒`

**Stack**  `…`

**Description**  The index is an unsigned byte that must be an index into the local variable array of the current frame (§3.6). The value on the top of the operand stack must be of type int. It is popped from the operand stack, and the value of the local variable at index is set to value.

**Notes**  The istore opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
**istore_<n>**

**Operation**  Store int into local variable

**Format**  

```
istore_<n>
```

**Forms**

- `istore_0 = 59 (0x3b)`
- `istore_1 = 60 (0x3c)`
- `istore_2 = 61 (0x3d)`
- `istore_3 = 62 (0x3e)`

**Operand**  ...

**Stack**  ...

**Description**  

The `<n>` must be an index into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type int. It is popped from the operand stack, and the value of the local variable at `<n>` is set to `value`.

**Notes**  

Each of the `istore_<n>` instructions is the same as `istore` with an index of `<n>`, except that the operand `<n>` is implicit.
**Operation**  Subtract int

**Format**

\[
\text{isub}
\]

**Forms**

\[\text{isub} = 100 \ (0x64)\]

**Operand Stack** ...., value1, value2 ⇒ ...., result

**Description**  Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 − value2. The result is pushed onto the operand stack.

For int subtraction, a − b produces the same result as a + (−b). For int values, subtraction from zero is the same as negation.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type int. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical difference of the two values.

Despite the fact that overflow may occur, execution of an isub instruction never throws a runtime exception.
### iushr

**Operation**  Logical shift right int

**Format**  

```
| iushr |
```

**Forms**  

\[ iushr = 124 \ (0x7c) \]

**Operand**  

\[ \ldots, \text{value1}, \text{value2} \Rightarrow \]

**Stack**  

\[ \ldots, \text{result} \]

**Description**  

Both \text{value1} and \text{value2} must be of type int. The values are popped from the operand stack. An int result is calculated by shifting \text{value1} right by \( s \) bit positions, with zero extension, where \( s \) is the value of the low 5 bits of \text{value2}. The result is pushed onto the operand stack.

**Notes**  

If \text{value1} is positive and \( s \) is \text{value2} \& 0x1f, the result is the same as that of \text{value1} \( \gg s \); if \text{value1} is negative, the result is equal to the value of the expression \((\text{value1} \gg s) + (2 << \sim s)\). The addition of the \((2 << \sim s)\) term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 31, inclusive.
**ixor**

**Operation**  
Boolean XOR `int`  

**Format**  
```
ixor
```

**Forms**  
`ixor = 130 (0x82)`  

**Operand**  
`. . . , value1, value2 ⇒`  

**Stack**  
`. . . , result`

**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 exclusive OR of `value1` and `value2`. The `result` is pushed onto the operand stack.
**jsr**

**Operation**  Jump subroutine

**Format**

```
| jsr      |
| branchbyte1 |
| branchbyte2 |
```

**Forms**  
\[ jsr = 168 \text{ (0xa8)} \]

**Operand**  
... ⇒

**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 \((\text{branchbyte1} \ll 8) \mid \text{branchbyte2} \). 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 clauses of the Java programming language (see §7.13, “Compiling finally”). Note that jsr pushes the address onto the operand stack and ret gets it out of a local variable. This asymmetry is intentional.
**jsr_w**

**Operation**
Jump subroutine (wide index)

**Format**

<table>
<thead>
<tr>
<th>jsr_w</th>
<th>branchbyte1</th>
<th>branchbyte2</th>
<th>branchbyte3</th>
<th>branchbyte4</th>
</tr>
</thead>
</table>

**Forms**

\( jsr_w = 201 \) (0xc9)

**Operand Stack**

\( \ldots \Rightarrow \ldots, \text{address} \)

**Description**
The address of the opcode of the instruction immediately following this \( jsr_w \) instruction is pushed onto the operand stack as a value of type `returnAddress`. The unsigned `branchbyte1`, `branchbyte2`, `branchbyte3`, and `branchbyte4` are used to construct a signed 32-bit offset, where the offset is \((\text{branchbyte1} \ll 24) | (\text{branchbyte2} \ll 16) | (\text{branchbyte3} \ll 8) | \text{branchbyte4}\). Execution proceeds at that offset from the address of this \( jsr_w \) instruction. The target address must be that of an opcode of an instruction within the method that contains this \( jsr_w \) instruction.

**Notes**
The \( jsr_w \) instruction is used with the `ret` instruction in the implementation of the `finally` clauses of the Java programming language (see §7.13, “Compiling `finally`”). Note that \( jsr_w \) pushes the address onto the operand stack and `ret` gets it out of a local variable. This asymmetry is intentional.

Although the \( jsr_w \) instruction takes a 4-byte branch offset, other factors limit the size of a method to 65535 bytes (§4.11). This limit may be raised in a future release of the Java virtual machine.
$l2d$

**Operation**  
Convert long to double

**Format**  

```
| l2d |
```

**Forms**  
$l2d = 138 (0x8a)$

**Operand**  
..., value ⇒

**Stack**  
..., result

**Description**  
The value on the top of the operand stack must be of type long. It is popped from the operand stack and converted to a double result using IEEE 754 round to nearest mode. The result is pushed onto the operand stack.

**Notes**  
The $l2d$ instruction performs a widening primitive conversion (JLS3 §5.1.2) that may lose precision because values of type double have only 53 significand bits.
### l2f

#### Operation
Convert `long` to `float`

#### Format
```
| l2f |
```

#### Forms
`l2f = 137 (0x89)`

#### Operand
`... , value ⇒`

#### Stack
`... , result`

#### Description
The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to a `float` result using IEEE 754 round to nearest mode. The `result` is pushed onto the operand stack.

#### Notes
The `l2f` instruction performs a widening primitive conversion (JLS3 §5.1.2) that may lose precision because values of type `float` have only 24 significand bits.
**l2i**

**Operation** Convert `long` to `int`

**Format**

```
l2i
```

**Forms**

`l2i = 136 (0x88)`

**Operand Stack**

```
..., value ⇒
```

**Description**

The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack and converted to an `int result` by taking the low-order 32 bits of the `long` value and discarding the high-order 32 bits. The `result` is pushed onto the operand stack.

**Notes**

The `l2i` instruction performs a narrowing primitive conversion (JLS3 §5.1.3). It may lose information about the overall magnitude of `value`. The `result` may also not have the same sign as `value`. 
*ladd*  

**Operation**  
Add *long*

**Format**  

```
| ladd |
```

**Forms**  
`ladd = 97 (0x61)`

**Operand**  
`..., value1, value2 ⇒`

**Stack**  
`..., result`

**Description**  
Both `value1` and `value2` must be of type `long`. The values are popped from the operand stack. The `long` result is `value1 + value2`. The result is pushed onto the operand stack.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type `long`. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an `ladd` instruction never throws a runtime exception.
**laload**

**Operation**  
Load long from array

**Format**  

```
laload
```

**Forms**  

\[ laload = 47 \ (0x2f) \]

**Operand Stack**  

\[ \ldots, \text{arrayref}, \text{index} \Rightarrow \ldots, \text{value} \]

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

**Runtime Exceptions**  
If `arrayref` is null, `laload` throws a `NullPointerException`. Otherwise, if `index` is not within the bounds of the array referenced by `arrayref`, the `laload` instruction throws an `ArrayIndexOutOfBoundsException`.
**land**

**Operation**  Boolean AND long

**Format**  

```
land
```

**Forms**  

$land = 127 \ (0x7f)$

**Operand**  

\[\ldots, value1, value2 \Rightarrow\]

**Stack**  

\[\ldots, result\]

**Description**  

Both $value1$ and $value2$ must be of type long. They are popped from the operand stack. A long result is calculated by taking the bitwise AND of $value1$ and $value2$. The result is pushed onto the operand stack.
lastore

Operation  Store into long array

Format  

Forms  lastore = 80 (0x50)

Operand  ... , arrayref, index, value ⇒

Stack  ...

Description  The arrayref must be of type reference and must refer to an array whose components are of type long. The index must be of type int, and value must be of type long. The arrayref, index, and value are popped from the operand stack. The long value is stored as the component of the array indexed by index.

Runtime Exceptions  If arrayref is null, lastore throws a NullPointerException. Otherwise, if index is not within the bounds of the array referenced by arrayref, the lastore instruction throws an ArrayIndexOutOfBoundsException.
**lcmp**

**Operation**  Compare `long`

**Format**  

```
lcmp
```

**Forms**  

\[ lcmp = 148 \ (0x94) \]

**Operand**  

\[ \ldots, \text{value1}, \text{value2} \Rightarrow \]

**Stack**  

\[ \ldots, \text{result} \]

**Description**  Both `value1` and `value2` must be of type `long`. They are both popped from the operand stack, and a signed integer comparison is performed. If `value1` is greater than `value2`, the `int` value 1 is pushed onto the operand stack. If `value1` is equal to `value2`, the `int` value 0 is pushed onto the operand stack. If `value1` is less than `value2`, the `int` value –1 is pushed onto the operand stack.
### lconst_<l>

**Operation**  Push long constant

**Format**  

<table>
<thead>
<tr>
<th>lconst_&lt;l&gt;</th>
</tr>
</thead>
</table>

**Forms**  

- `lconst_0 = 9 (0x9)`
- `lconst_1 = 10 (0xa)`

**Operand**  

| ... ⇒ |

**Stack**  

| ..., <l> |

**Description**  Push the long constant <l> (0 or 1) onto the operand stack.
**ldc**

**Operation**
Push item from runtime constant pool

**Format**

```
   ldc
   index
```

**Forms**

```
ldc = 18 (0x12)
```

**Operand**

```
... ⇒
```

**Stack**
```
..., value
```

**Description**
The *index* is an unsigned byte that must be a valid index into the runtime constant pool of the current class (§3.6). The runtime constant pool entry at *index* either must be a runtime constant of type **int** or **float**, or must be a symbolic reference to a class (§5.4.3.1) or a string literal (§5.1).

If the runtime constant pool entry is a runtime constant of type **int** or **float**, the numeric *value* of that runtime constant is pushed onto the operand stack as an **int** or **float**, respectively.

Otherwise, if the runtime constant pool entry is a reference to an instance of class **String** representing a string literal (§5.1), then a reference to that instance, *value*, is pushed onto the operand stack.

Otherwise, the runtime constant pool entry must be a symbolic reference to a class (§4.4.1). The named class is resolved (§5.4.3.1) and a reference to the Class object representing that class, *value*, is pushed onto the operand stack.

**Linking Exceptions**
During resolution of the symbolic reference to the class, any of the exceptions pertaining to class resolution documented in §5.4.3.1 can be thrown.

**Notes**
The *ldc* instruction can only be used to push a value of type **float** taken from the float value set (§3.3.2) because a constant of type **float** in the constant pool (§4.4.4) must be taken from the float value set.
**ldc_w**

**Operation**
Push item from runtime constant pool (wide index)

**Format**

<table>
<thead>
<tr>
<th>ldc_w</th>
<th>indexbyte1</th>
<th>indexbyte2</th>
</tr>
</thead>
</table>

**Forms**

ldc_w = 19 (0x13)

**Operand**

... ⇒

**Stack**

..., value

**Description**
The unsigned indexbyte1 and indexbyte2 are assembled into an unsigned 16-bit index into the runtime constant pool of the current class (§3.6), where the value of the index is calculated as (indexbyte1 << 8) | indexbyte2. The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index either must be a runtime constant of type int or float, or must be a symbolic reference to a class (§5.4.3.1) or a string literal (§5.1).

If the runtime constant pool entry is a runtime constant of type int or float, the numeric value of that runtime constant is pushed onto the operand stack as an int or float, respectively.

Otherwise, if the runtime constant pool entry is a reference to an instance of class String representing a string literal (§5.1), then a reference to that instance, value, is pushed onto the operand stack.

Otherwise, the runtime constant pool entry must be a symbolic reference to a class (§4.4.1). The named class is resolved (§5.4.3.1) and a reference to the Class object representing that class, value, is pushed onto the operand stack.

**Linking**

**Exceptions**

During resolution of the symbolic reference to the class, any of the exceptions pertaining to class resolution documented in §5.4.3.1 can be thrown.
Notes

The \textit{ldc\_w} instruction is identical to the \textit{ldc} instruction except for its wider runtime constant pool index.

The \textit{ldc\_w} instruction can only be used to push a value of type \texttt{float} taken from the float value set (§3.3.2) because a constant of type \texttt{float} in the constant pool (§4.4.4) must be taken from the float value set.
**ldc2_w**  

**Operation**  
Push `long` or `double` from runtime constant pool (wide index)

**Format**

<table>
<thead>
<tr>
<th>ldc2_w</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**  
`ldc2_w` = 20 (0x14)

**Operand**  
... ⇒

**Stack**  
..., value

**Description**  
The unsigned `indexbyte1` and `indexbyte2` are assembled into an unsigned 16-bit index into the runtime constant pool of the current class (§3.6), where the value of the index is calculated as `(indexbyte1 << 8) | indexbyte2`. The index must be a valid index into the runtime constant pool of the current class. The runtime constant pool entry at the index must be a runtime constant of type `long` or `double` (§5.1). The numeric value of that runtime constant is pushed onto the operand stack as a `long` or `double`, respectively.

**Notes**  
Only a wide-index version of the `ldc2_w` instruction exists; there is no `ldc2` instruction that pushes a `long` or `double` with a single-byte index.

The `ldc2_w` instruction can only be used to push a value of type `double` taken from the double value set (§3.3.2) because a constant of type `double` in the constant pool (§4.4.5) must be taken from the double value set.
ldiv

**Operation**  Divide long

**Format**  

| ldiv |

**Forms**  ldiv = 109 (0x6d)

**Operand Stack**  ..., value1, value2 ⇒

**Stack**  ..., result

**Description**  Both value1 and value2 must be of type long. The values are popped from the operand stack. The long result is the value of the Java programming language expression value1 / value2. The result is pushed onto the operand stack.

A long division rounds towards 0; that is, the quotient produced for long values in \( n / d \) is a long value \( q \) whose magnitude is as large as possible while satisfying \(|d \cdot q| \leq |n|\). Moreover, \( q \) is positive when \(|n| \geq |d| \) and \( n \) and \( d \) have the same sign, but \( q \) is negative when \(|n| \geq |d| \) 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 largest possible magnitude for the long type and the divisor is \(-1\), then overflow occurs and the result is equal to the dividend; despite the overflow, no exception is thrown in this case.

**Runtime Exception**  If the value of the divisor in a long division is 0, ldiv throws an ArithmeticException.
**lload**

**Operation**  Load long from local variable

**Format**

<table>
<thead>
<tr>
<th>lload</th>
<th>index</th>
</tr>
</thead>
</table>

**Forms**  \( lload = 22 \ (0x16) \)

**Operand**  \( \ldots \Rightarrow \)

**Stack**  \( \ldots, value \)

**Description**  The index is an unsigned byte. Both index and index + 1 must be indices into the local variable array of the current frame (§3.6). The local variable at index must contain a long. The value of the local variable at index is pushed onto the operand stack.

**Notes**  The lload opcode can be used in conjunction with the wide instruction to access a local variable using a two-byte unsigned index.
\textit{lload_\(<n>\)}

Operation \quad \text{Load long from local variable}

Format \quad \textit{lload_\(<n>\)}

Forms \quad \begin{align*}
  lload_0 &= 30 (0x1e) \\
  lload_1 &= 31 (0x1f) \\
  lload_2 &= 32 (0x20) \\
  lload_3 &= 33 (0x21)
\end{align*}

Operand Stack \quad \ldots \Rightarrow \ldots, \textit{value}

Description \quad \text{Both <n> and <n> + 1 must be indices into the local variable array of the current frame (§3.6). The local variable at <n> must contain a long. The value of the local variable at <n> is pushed onto the operand stack.}

Notes \quad \text{Each of the lload_\(<n>\) instructions is the same as lload with an index of <n>, except that the operand <n> is implicit.}
**lmul**

**Operation**  Multiply long

**Format**  

```
  lmul
```

**Forms**  

```
lmul = 105 (0x69)
```

**Operand Stack**  

```
..., value1, value2 ⇒ ...
```

**Description**  

Both `value1` and `value2` must be of type `long`. The values are popped from the operand stack. The `long` result is `value1 * value2`. The result is pushed onto the operand stack.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type `long`. If overflow occurs, the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an `lmul` instruction never throws a runtime exception.
**Ineg**

**Operation**  Negate long

**Format**  
```
Ineg
```

**Forms**  
\( \text{lneg} = 117 \ (0x75) \)

**Operand**  
\( \cdots, \text{value} \Rightarrow \)

**Stack**  
\( \cdots, \text{result} \)

**Description**  
The value must be of type long. It is popped from the operand stack. The long result is the arithmetic negation of value, \(-value\). The result is pushed onto the operand stack.

For long values, negation is the same as subtraction from zero. Because the Java 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 long results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all long values \( x \), \(-x\) equals \((\sim x) + 1\).
**lookupswitch**

**Operation**
Access jump table by key match and jump

**Format**

<table>
<thead>
<tr>
<th>lookupswitch</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;0-3 byte pad&gt;</td>
</tr>
<tr>
<td>defaultbyte1</td>
</tr>
<tr>
<td>defaultbyte2</td>
</tr>
<tr>
<td>defaultbyte3</td>
</tr>
<tr>
<td>defaultbyte4</td>
</tr>
<tr>
<td>npairs1</td>
</tr>
<tr>
<td>npairs2</td>
</tr>
<tr>
<td>npairs3</td>
</tr>
<tr>
<td>npairs4</td>
</tr>
<tr>
<td>match-offset pairs</td>
</tr>
</tbody>
</table>

**Forms**

*lookupswitch = 171 (0xab)*

**Operand**

...,
\( key \Rightarrow \)

**Stack**

...

**Description**

A *lookupswitch* is a variable-length instruction. Immediately after the *lookupswitch* opcode, between zero and three bytes must act as padding, such that *defaultbyte1* begins at an address that is a multiple of four bytes from the start of the current method (the opcode of its first instruction). Immediately after the padding follow a series of signed 32-bit values: *default*, *npairs*, and then *npairs* pairs of signed 32-bit values. The *npairs* must be greater than or equal to 0. Each of the *npairs* pairs consists of an int *match* and a signed 32-bit *offset*. Each of these signed 32-bit values is constructed from four unsigned bytes as \((byte1 \ll 24) \mid (byte2 \ll 16) \mid (byte3 \ll 8) \mid byte4\).
lookupswitch (cont.)  lookupswitch (cont.)

The table match-offset pairs of the lookupswitch instruction must be sorted in increasing numerical order by match.

The key must be of type int and is popped from the operand stack. The key is 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 lookupswitch 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 lookupswitch 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 lookupswitch instruction.

Notes

The alignment required of the 4-byte operands of the lookupswitch instruction guarantees 4-byte alignment of those operands if and only if the method that contains the lookupswitch is positioned on a 4-byte boundary.

The match-offset pairs are sorted to support lookup routines that are quicker than linear search.
**lor**

**Operation**  Boolean OR long

**Format**  

| lor |

**Forms**  

lor = 129 (0x81)

**Operand**  

..., value1, value2 ⇒

**Stack**  

..., result

**Description**  Both value1 and value2 must be of type long. They are popped from the operand stack. A long result is calculated by taking the bitwise inclusive OR of value1 and value2. The result is pushed onto the operand stack.
**lrem**

**Operation**  
Remainder `long`

**Format**  
```
  $lrem$
```

**Forms**  
$lrem = 113 \ (0x71)$

**Operand**  
$\ldots, value1, value2 \Rightarrow$

**Stack**  
$\ldots, result$

**Description**  
Both $value1$ and $value2$ must be of type `long`. The values are popped from the operand stack. The `long` result is $value1 - (value1 / value2) * value2$. The result is pushed onto the operand stack.

The result of the `lrem` instruction is such that $(a/b) * b + (a \% b)$ is equal to $a$. This identity holds even in the special case in which the dividend is the negative `long` 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 `long` remainder operator is 0, `lrem` throws an `ArithmeticException`.
**Ireturn**

**Operation**  Return long from method

**Format**  

|  
|---|
| **Ireturn**  

**Forms**  

\( Ireturn = 173 \) (0xad)

**Operand**  

\( \ldots, value \Rightarrow \)

**Stack**  

[empty]

**Description**  
The current method must have return type long. The value must be of type long. If the current method is a synchronized method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a monitorexit instruction in the current thread. If no exception is thrown, value is popped from the operand stack of the current frame (§3.6) 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 interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**  

If the virtual machine implementation does not enforce the rules on structured locking described in §3.11.10, then if the current method is a synchronized method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, Ireturn throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitoreenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the first of those rules is violated during invocation of the current method, then Ireturn throws an IllegalMonitorStateException.
**lshl**

**Operation**  
Shift left long

**Format**  
```
lshl
```

**Forms**  
lshl = 121 (0x79)

**Operand**  
..., value1, value2 ⇒

**Stack**  
..., result

**Description**  
The value1 must be of type long, and value2 must be of type int. The values are popped from the operand stack. A long result is calculated by shifting value1 left by s bit positions, where s is the low 6 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 therefore always in the range 0 to 63, inclusive, as if value2 were subjected to a bitwise logical AND with the mask value 0x3f.
**lshr**

**Operation**  
Arithmetic shift right `long`

**Format**  
```
lshr
```

**Forms**  
`lshr = 123 (0x7b)`

**Operand**  
`..., value1, value2 ⇒`

**Stack**  
`..., result`

**Description**  
The `value1` must be of type `long`, and `value2` must be of type `int`. The values are popped from the operand stack. A `long` result is calculated by shifting `value1` right by `s` bit positions, with sign extension, where `s` is the value of the low 6 bits of `value2`. The result is pushed onto the operand stack.

**Notes**  
The resulting value is `\left\lfloor \frac{value1}{2^s} \right\rfloor`, where `s` is `value2 & 0x3f`. For non-negative `value1`, this is equivalent to truncating `long` division by 2 to the power `s`. The shift distance actually used is therefore always in the range 0 to 63, inclusive, as if `value2` were subjected to a bitwise logical AND with the mask value 0x3f.
**Istore**

**Operation**  
Store `long` into local variable

**Format**

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Istore</strong></td>
<td></td>
</tr>
<tr>
<td><strong>index</strong></td>
<td></td>
</tr>
</tbody>
</table>
```

**Forms**

`Istore = 55 (0x37)`

**Operand**  
`...`, `value` ⇒

**Stack**  
`...`

**Description**  
The `index` is an unsigned byte. Both `index` and `index + 1` must be indices into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at `index` and `index + 1` are set to `value`.

**Notes**  
The `Istore` opcode can be used in conjunction with the `wide` instruction to access a local variable using a two-byte unsigned index.
**lstore_<n>**

**Operation**  Store long into local variable

**Format**

```
 lstore_<n>
```

**Forms**

- `lstore_0 = 63 (0x3f)`
- `lstore_1 = 64 (0x40)`
- `lstore_2 = 65 (0x41)`
- `lstore_3 = 66 (0x42)`

**Operand Stack**

```
 ..., value ⇒ ...
```

**Description**  Both `<n>` and `<n> + 1` must be indices into the local variable array of the current frame (§3.6). The `value` on the top of the operand stack must be of type `long`. It is popped from the operand stack, and the local variables at `<n>` and `<n> + 1` are set to `value`.

**Notes**  Each of the `lstore_<n>` instructions is the same as `lstore` with an index of `<n>`, except that the operand `<n>` is implicit.
**lsub**

**Operation**  Subtract long

**Format**  

```
   lsub
```

**Forms**  
lsub = 101 (0x65)

**Operand Stack**  

```
..., value1, value2 ⇒
```

**Description**  
Both value1 and value2 must be of type long. The values are popped from the operand stack. The long result is value1 − value2. The result is pushed onto the operand stack.

For long subtraction, a−b produces the same result as a+(−b). For long values, subtraction from zero is the same as negation.

The result is the 64 low-order bits of the true mathematical result in a sufficiently wide two’s-complement format, represented as a value of type long. 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.

Despite the fact that overflow may occur, execution of an lsub instruction never throws a runtime exception.
**lushr**

**Operation** Logical shift right `long`

**Format**

```
lushr
```

**Forms**

`lushr = 125 (0x7d)`

**Operand** `..., value1, value2 ⇒`

**Stack** `..., result`

**Description** The `value1` must be of type `long`, and `value2` must be of type `int`. The values are popped from the operand stack. A `long` result is calculated by shifting `value1` right logically (with zero extension) by the amount indicated by the low 6 bits of `value2`. The result is pushed onto the operand stack.

**Notes** If `value1` is positive and `s` is `value2 & 0x3f`, 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) + (2L << ~s)`. The addition of the `(2L << ~s)` term cancels out the propagated sign bit. The shift distance actually used is always in the range 0 to 63, inclusive.
**lxor**

**Operation**
Boolean XOR long

**Format**
```
   lxor
```

**Forms**
lxor = 131 (0x83)

**Operand**
... value1, value2 ⇒

**Stack**
... result

**Description**
Both value1 and value2 must be of type long. They are popped from the operand stack. A long result is calculated by taking the bitwise exclusive OR of value1 and value2. The result is pushed onto the operand stack.
**monitorenter**  

**Operation** Enter monitor for object

**Format**

```
monitorenter
```

**Forms**

`monitorenter = 194 (0xc2)`

**Operand** .., `objectref` ⇒

**Stack** …

**Description** The `objectref` must be of type `reference`.

Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes `monitorenter` attempts to gain ownership of the monitor associated with `objectref`, as follows.

If the entry count of the monitor associated with `objectref` is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.

If the thread already owns the monitor associated with `objectref`, it reenters the monitor, incrementing its entry count.

If another thread already owns the monitor associated with `objectref`, the thread blocks until the monitor’s entry count is zero, then tries again to gain ownership.

**Runtime Exception**

If `objectref` is `null`, `monitorenter` throws a `NullPointerException`.

**Notes** A `monitorenter` instruction may be used with one or more `monitorexit` instructions to implement a `synchronized` statement in the Java programming language (§7.15). The `monitorenter` and `monitorexit` instructions are not used in the implementation of synchronized methods, although they can be used to provide equivalent locking semantics. Monitor entry on invocation of a `synchronized` method, and monitor exit on its return, are handled implicitly by the Java virtual machine’s method invocation and return instructions, as if `monitorenter` and `monitorexit` were used.
The association of a monitor with an object may be managed in various ways that are beyond the scope of this specification. For instance, the monitor may be allocated and deallocated at the same time as the object. Alternatively, it may be dynamically allocated at the time when a thread attempts to gain exclusive access to the object and freed at some later time when no thread remains in the monitor for the object.

The synchronization constructs of the Java programming language require support for operations on monitors besides entry and exit. These include waiting on a monitor (Object.wait) and notifying other threads waiting on a monitor (Object.notifyAll and Object.notify). These operations are supported in the standard package java.lang supplied with the Java virtual machine. No explicit support for these operations appears in the instruction set of the Java virtual machine.
**monitorexit**

**Operation**
Exit monitor for object

**Format**

```
monitorexit
```

**Forms**

```
monitorexit = 195 (0xc3)
```

**Operand**

```
..., objectref ⇒
```

**Stack**

```
...
```

**Description**
The `objectref` must be of type `reference`.

The thread that executes `monitorexit` must be the owner of the monitor associated with the instance referenced by `objectref`.

The thread decrements the entry count of the monitor associated with `objectref`. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

**Runtime Exceptions**

If `objectref` is `null`, `monitorexit` throws a `NullPointerException`.

Otherwise, if the thread that executes `monitorexit` is not the owner of the monitor associated with the instance referenced by `objectref`, `monitorexit` throws an `IllegalMonitorStateException`.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the second of those rules is violated by the execution of this `monitorexit` instruction, then `monitorexit` throws an `IllegalMonitorStateException`.

**Notes**

One or more `monitorexit` instructions may be used with a `monitorenter` instruction to implement a synchronized statement in the Java programming language (§7.15). The `monitorenter` and `monitorexit` instructions are not used in the implementation of synchronized methods, although they can be used to provide equivalent locking semantics.
The Java virtual machine supports exceptions thrown within synchronized methods and synchronized statements differently:

- Monitor exit on normal synchronized method completion is handled by the Java virtual machine’s return instructions. Monitor exit on abrupt synchronized method completion is handled implicitly by the Java virtual machine’s `athrow` instruction.

- When an exception is thrown from within a synchronized statement, exit from the monitor entered prior to the execution of the synchronized statement is achieved using the Java virtual machine’s exception handling mechanism (§7.15).
**multianewarray**

**Operation**  
Create new multidimensional array

**Format**

```
multianewarray
indexbyte1
indexbyte2
dimensions
```

**Forms**  
`multianewarray = 197 (0xc5)`

**Operand**  
`, count1, [count2, ...] ⇒`

**Stack**  
`, arrayref`

**Description**  
The `dimensions` operand is an unsigned byte that must be greater than or equal to 1. It represents the number of dimensions of the array to be created. The operand stack must contain `dimensions` values. Each such value represents the number of components in a dimension of the array to be created, must be of type `int`, and must be nonnegative. The `count1` is the desired length in the first dimension, `count2` in the second, etc.

All of the `count` values are popped off the operand stack. The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is `(indexbyte1 << 8) | indexbyte2`. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved (§5.4.3.1). The resulting entry must be an array class type of dimensionality greater than or equal to `dimensions`. 
**multianewarray (cont.)**

A new multidimensional array of the array type is allocated from the garbage-collected heap. If any `count` value is zero, no subsequent dimensions are allocated. The components of the array in the first dimension are initialized to subarrays of the type of the second dimension, and so on. The components of the last allocated dimension of the array are initialized to the default initial value for the type of the components (JLS3 §4.12.5). A reference `arrayref` to the new array is pushed onto the operand stack.

**Linking Exceptions**

During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

Otherwise, if the current class does not have permission to access the element type of the resolved array class, `multianewarray` throws an `IllegalArgumentException`.

**Runtime Exception**

Otherwise, if any of the `dimensions` values on the operand stack are less than zero, the `multianewarray` instruction throws a `NegativeArraySizeException`.

**Notes**

It may be more efficient to use `newarray` or `anewarray` when creating an array of a single dimension.

The array class referenced via the runtime constant pool may have more dimensions than the `dimensions` operand of the `multianewarray` instruction. In that case, only the first `dimensions` of the dimensions of the array are created.
new

Operation Create new object

Format

<table>
<thead>
<tr>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

Forms \(\text{new} = 187 \ (0xbb)\)

Operand Stack \(\ldots \Rightarrow \ldots, \text{objectref}\)

Description The unsigned \(\text{indexbyte1}\) and \(\text{indexbyte2}\) are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is \((\text{indexbyte1} \ll 8) \mid \text{indexbyte2}\). The runtime constant pool item at the index must be a symbolic reference to a class or interface type. The named class or interface type is resolved (§5.4.3.1) and should result in a class type. Memory for a new instance of that class is allocated from the garbage-collected heap, and the instance variables of the new object are initialized to their default initial values (JLS3 §4.12.5). The \text{objectref}, a reference to the instance, is pushed onto the operand stack.

On successful resolution of the class, it is initialized (§5.5) if it has not already been initialized.

Linking

Exceptions During resolution of the symbolic reference to the class, array, or interface type, any of the exceptions documented in §5.4.3.1 can be thrown.

Otherwise, if the symbolic reference to the class, array, or interface type resolves to an interface or is an \text{abstract} class, \text{new} throws an \text{InstantiationError}. 
new (cont.)

Runtime Exception

Otherwise, if execution of this new instruction causes initialization of the referenced class, new may throw an Error as detailed in JLS3 §15.9.4.

Note

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.
newarray

Operation    Create new array

Format

\[
\begin{array}{c}
\text{newarray} \\
\text{atype}
\end{array}
\]

Forms    newarray = 188 (0xbc)

Operand    \ldots, count ⇒

Stack    \ldots, arrayref

Description    The count must be of type int. It is popped off the operand stack. The count represents the number of elements in the array to be created.

The atype is a code that indicates the type of array to create. It must take one of the following values:

<table>
<thead>
<tr>
<th>Array Type</th>
<th>atype</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_BOOLEAN</td>
<td>4</td>
</tr>
<tr>
<td>T_CHAR</td>
<td>5</td>
</tr>
<tr>
<td>T_FLOAT</td>
<td>6</td>
</tr>
<tr>
<td>T_DOUBLE</td>
<td>7</td>
</tr>
<tr>
<td>T_BYTE</td>
<td>8</td>
</tr>
<tr>
<td>T_SHORT</td>
<td>9</td>
</tr>
<tr>
<td>T_INT</td>
<td>10</td>
</tr>
<tr>
<td>T_LONG</td>
<td>11</td>
</tr>
</tbody>
</table>

A new array whose components are of type atype and of length count is allocated from the garbage-collected heap. A reference arrayref to this new array object is pushed into the operand stack. Each of the elements of the new array is initialized to the default initial value for the type of the array (JLS3 §4.12.5).
newarray (cont.)

Runtime Exception

If `count` is less than zero, `newarray` throws a `NegativeArraySizeException`.

Notes

In Sun’s implementation of the Java virtual machine, arrays of type `boolean` (`atype` is `T_BOOLEAN`) are stored as arrays of 8-bit values and are manipulated using the `baload` and `bastore` instructions, instructions that also access arrays of type `byte`. Other implementations may implement packed `boolean` arrays; the `baload` and `bastore` instructions must still be used to access those arrays.
**nop**

<table>
<thead>
<tr>
<th><strong>Operation</strong></th>
<th>Do nothing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format</strong></td>
<td><em>nop</em></td>
</tr>
<tr>
<td><strong>Forms</strong></td>
<td><em>nop</em> = 0 (0x0)</td>
</tr>
<tr>
<td><strong>Operand</strong></td>
<td>No change</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Do nothing.</td>
</tr>
</tbody>
</table>


**Operation**

Pop the top operand stack value

**Format**

```
pop
```

**Forms**

`pop = 87 (0x57)`

**Operand**

`…, value ⇒`

**Stack**

`…`

**Description**

Pop the top value from the operand stack.

The `pop` instruction must not be used unless `value` is a value of a category 1 computational type (§3.11.1).
**pop2**

**Operation**  
Pop the top one or two operand stack values

**Format**  
```
pop2
```

**Forms**  
`pop2 = 88 (0x58)`

**Operand Stack**

Form 1:

```
..., value2, value1 ⇒
...
```

where each of `value1` and `value2` is a value of a category 1 computational type (§3.11.1).

Form 2:

```
..., value ⇒
...
```

where `value` is a value of a category 2 computational type (§3.11.1).

**Description**  
Pop the top one or two values from the operand stack.
**putfield**  

**Operation**  
Set field in object

**Format**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>putfield</strong></td>
<td><strong>indexbyte1</strong></td>
<td><strong>indexbyte2</strong></td>
</tr>
</tbody>
</table>

**Forms**  
\( \text{putfield} = 181 \) (0xb5)

**Operand**  
..., objectref, value \( \Rightarrow \)

**Stack**  
...  

**Description**  
The unsigned \( \text{indexbyte1} \) and \( \text{indexbyte2} \) are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is \( (\text{indexbyte1} \ll 8) \mid \text{indexbyte2} \). The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class in which the field is to be found. The class of objectref must not be an array. If the field is protected (§4.6), and it is a member of a superclass of the current class, and the field is not declared in the same runtime package (§5.3) as the current class, then the class of objectref must be either the current class or a subclass of the current class.

The referenced field is resolved (§5.4.3.3). The type of a value stored by a putfield instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is boolean, byte, char, short, or int, then the value must be an int. If the field descriptor type is float, long, or double, then the value must be a float, long, or double, respectively. If the field descriptor type is a reference type, then the value must be of a type that is assignment compatible (JLS3 §5.2) with the field descriptor type. If the field is final, it must be declared in the current class, and the instruction must occur in an instance initialization method (\(<\text{init}>\)) of the current class.
putfield (cont.)

The value and objectref are popped from the operand stack. The objectref must be of type reference. The value undergoes value set conversion (§3.8.3), resulting in value’, and the referenced field in objectref is set to value’.

Linking Exceptions

During resolution of the symbolic reference to the field, any of the exceptions pertaining to field resolution documented in §5.4.3.3 can be thrown.

Otherwise, if the resolved field is a static field, putfield throws an IncompatibleClassChangeError.

Otherwise, if the field is final, it must be declared in the current class, and the instruction must occur in an instance initialization method (<init>) of the current class. Otherwise, an IllegalAccessError is thrown.

Runtime Exception

Otherwise, if objectref is null, the putfield instruction throws a NullPointerException.
**putstatic**

**Operation**  
Set static field in class

**Format**

<table>
<thead>
<tr>
<th>putstatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexbyte1</td>
</tr>
<tr>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

**Forms**

putstatic = 179 (0xb3)

**Operand**  
..., value ⇒

**Stack**

...  

**Description**

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the runtime constant pool of the current class (§3.6), where the value of the index is (indexbyte1 << 8) | indexbyte2. The runtime constant pool item at that index must be a symbolic reference to a field (§5.1), which gives the name and descriptor of the field as well as a symbolic reference to the class or interface in which the field is to be found. The referenced field is resolved (§5.4.3.3).

On successful resolution of the field the class or interface that declared the resolved field is initialized (§5.5) if that class or interface has not already been initialized.

The type of a value stored by a putstatic instruction must be compatible with the descriptor of the referenced field (§4.3.2). If the field descriptor type is boolean, byte, char, short, or int, then the value must be an int. If the field descriptor type is float, long, or double, then the value must be a float, long, or double, respectively. If the field descriptor type is a reference type, then the value must be of a type that is assignment compatible (JLS3 §5.2) with the field descriptor type. If the field is final, it must be declared in the current class, and the instruction must occur in the <clinit> method of the current class.
**putstatic (cont.)**

The *value* is popped from the operand stack and undergoes value set conversion (§3.8.3), resulting in *value'*. The class field is set to *value'*.  

**Linking Exceptions**

During resolution of the symbolic reference to the class or interface field, any of the exceptions pertaining to field resolution documented in §5.4.3.3 can be thrown.

Otherwise, if the resolved field is not a `static` (class) field or an interface field, *putstatic* throws an `IncompatibleClassChangeError`.

Otherwise, if the field is `final`, it must be declared in the current class, and the instruction must occur in the `<clinit>` method of the current class. Otherwise, an `IllegalAccessError` is thrown.

**Runtime Exception**

Otherwise, if execution of this *putstatic* instruction causes initialization of the referenced class or interface, *putstatic* may throw an *Error* as detailed in §5.5.

**Notes**

A *putstatic* instruction may be used only to set the value of an interface field on the initialization of that field. Interface fields may be assigned to only once, on execution of an interface variable initialization expression when the interface is initialized (JLS3 §9.3.1, §5.5).
ret

Operation  Return from subroutine

Format

\[
\begin{array}{l}
\text{ret} \\
\text{index}
\end{array}
\]

Forms  \( ret = 169 \) (0xa9)

Operand Stack  No change

Description  The \textit{index} is an unsigned byte between 0 and 255, inclusive. The local variable at \textit{index} in the current frame (§3.6) must contain a value of type \textit{returnAddress}. The contents of the local variable are written into the Java virtual machine’s pc register, and execution continues there.

Notes  The \textit{ret} instruction is used with \textit{jsr} or \textit{jsr}_{\text{w}} instructions in the implementation of the \texttt{finally} clauses of the Java programming language (see §7.13, “Compiling \texttt{finally}”). Note that \textit{jsr} pushes the address onto the operand stack and \textit{ret} gets it out of a local variable. This asymmetry is intentional.

The \textit{ret} instruction should not be confused with the \textit{return} instruction. A \textit{return} instruction returns control from a method to its invoker, without passing any value back to the invoker.

The \textit{ret} opcode can be used in conjunction with the \textit{wide} instruction to access a local variable using a two-byte unsigned index.
**return**

**Operation**
Return void from method

**Format**
```
return
```

**Forms**
return = 177 (0xb1)

**Operand Stack**
... ⇒ [empty]

**Description**
The current method must have return type void. If the current method is a synchronized method, the monitor entered or reentered on invocation of the method is updated and possibly exited as if by execution of a monitor-exit instruction in the current thread. If no exception is thrown, any values on the operand stack of the current frame (§3.6) are discarded.

The interpreter then returns control to the invoker of the method, reinstating the frame of the invoker.

**Runtime Exceptions**
If the virtual machine implementation does not enforce the rules on structured locking described in §3.11.10, then if the current method is a synchronized method and the current thread is not the owner of the monitor entered or reentered on invocation of the method, return throws an IllegalMonitorStateException. This can happen, for example, if a synchronized method contains a monitorexit instruction, but no monitorenter instruction, on the object on which the method is synchronized.

Otherwise, if the virtual machine implementation enforces the rules on structured locking described in §3.11.10 and if the first of those rules is violated during invocation of the current method, then return throws an IllegalMonitorStateException.
**saload**

**Operation**  Load short from array

**Format**  

```
saload
```

**Forms**  

```
saload = 53 (0x35)
```

**Operand Stack**  

```
..., arrayref, index ⇒
```

**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 `int`. Both `arrayref` and `index` are popped from the operand stack. The component of the array at `index` is retrieved and sign-extended to an `int` value. That value is pushed onto 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`.

**Stack**  

```
..., value
```
**sastore**

**Operation**
Store into short array

**Format**

```
sastore
```

**Forms**

`sastore = 86 (0x56)`

**Operand Stack**

```
..., array, index, value =>
```

**Description**
The `arrayref` must be of type `reference` and must refer to an array whose components are of type `short`. Both `index` and `value` must be of type `int`. The `arrayref`, `index`, and `value` are popped from the operand stack. The `int value` is truncated to a `short` and stored as the component of the array indexed by `index`.

**Runtime Exceptions**

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`.
**sipush**

**Operation**  Push short

**Format**

<table>
<thead>
<tr>
<th>sipush</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte1</td>
</tr>
<tr>
<td>byte2</td>
</tr>
</tbody>
</table>

**Forms**  sipush = 17 (0x11)

**Operand Stack**  ...

**Description**  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 `value`. That `value` is pushed onto the operand stack.
Operation Swap the top two operand stack values

Format

Forms $\textit{swap} = 95 \ (0x5f)$

Operand $\ldots, \textit{value2}, \textit{value1} \Rightarrow$

Stack $\ldots, \textit{value1}, \textit{value2}$

Description Swap the top two values on the operand stack.

The $\textit{swap}$ instruction must not be used unless $\textit{value1}$ and $\textit{value2}$ are both values of a category 1 computational type (§3.11.1).

Notes The Java virtual machine does not provide an instruction implementing a swap on operands of category 2 computational types.
tables\texttt{witch} \quad \texttt{tables\texttt{witch}}

\textbf{Operation} \quad \text{Access jump table by index and jump}

\begin{center}
\begin{tabular}{|l|}
\hline
\texttt{tables\texttt{witch}} \\
\texttt{\textless 0-3 byte pad\textgreater} \\
defaultbyte1 \\
defaultbyte2 \\
defaultbyte3 \\
defaultbyte4 \\
lowbyte1 \\
lowbyte2 \\
lowbyte3 \\
lowbyte4 \\
highbyte1 \\
highbyte2 \\
highbyte3 \\
highbyte4 \\
jump offsets… \\
\hline
\end{tabular}
\end{center}

\textbf{Forms} \quad \texttt{tables\texttt{witch}} = 170 (0xaa)

\textbf{Operand} \quad \ldots, \textit{index} \Rightarrow

\textbf{Stack} \quad \ldots

\textbf{Description} \quad A \textit{tables\texttt{witch}} is a variable-length instruction. Immediately after the \textit{tables\texttt{witch}} opcode, between zero and three bytes must act as padding, such that \texttt{defaultbyte1} begins at an address that is a multiple of four bytes from the start of the current method (the opcode of its first instruction). Immediately after the padding are bytes constituting three signed 32-bit values: \texttt{default}, \texttt{low}, and \texttt{high}. Immediately following are bytes constituting a series of \texttt{high} − \texttt{low} + 1 signed 32-bit offsets. The value \texttt{low} must be less than or equal to \texttt{high}. The \texttt{high} − \texttt{low} + 1 signed 32-bit offsets are treated as a 0-based jump table. Each of these signed 32-bit values is constructed as \texttt{(byte1 ◊ ◊ ◊ 24)} | \texttt{(byte2 ◊ ◊ ◊ 16)} | \texttt{(byte3 ◊ ◊ ◊ ◊ 8)} | \texttt{byte4}.
tables\textit{switch} \textit{(cont.)}  \hspace{1cm} \textit{tables\textit{switch}} \textit{(cont.)}

The \textit{index} must be of type \texttt{int} and is popped from the operand stack. If \textit{index} is less than \textit{low} or \textit{index} is greater than \textit{high}, then a target address is calculated by adding \texttt{default} to the address of the opcode of this \textit{tables\textit{switch}} instruction. Otherwise, the offset at position \textit{index} – \textit{low} of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this \textit{tables\textit{switch}} instruction. Execution then continues at the target address.

The target address that can be calculated from each jump table offset, as well as the one that can be calculated from \texttt{default}, must be the address of an opcode of an instruction within the method that contains this \textit{tables\textit{switch}} instruction.

\textbf{Notes}  \hspace{1cm}  

The alignment required of the 4-byte operands of the \textit{tables\textit{switch}} instruction guarantees 4-byte alignment of those operands if and only if the method that contains the \textit{tables\textit{switch}} starts on a 4-byte boundary.
**wide**

**Operation**
Extend local variable index by additional bytes

**Format 1:**

<table>
<thead>
<tr>
<th>wide</th>
<th>&lt;opcode&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>indexbyte1</td>
</tr>
<tr>
<td></td>
<td>indexbyte2</td>
</tr>
</tbody>
</table>

where <opcode> is one of iload, fload, aload, iload, dload, astore, fstore, astore, lstore, dstore, or ret

**Format 2:**

<table>
<thead>
<tr>
<th>wide</th>
<th>iinc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>indexbyte1</td>
</tr>
<tr>
<td></td>
<td>indexbyte2</td>
</tr>
<tr>
<td></td>
<td>constbyte1</td>
</tr>
<tr>
<td></td>
<td>constbyte2</td>
</tr>
</tbody>
</table>

**Forms**
wide = 196 (0xc4)

**Operand Stack**
Same as modified instruction

**Description**
The wide instruction modifies the behavior of another instruction. It takes one of two formats, depending on the instruction being modified. The first form of the wide instruction modifies one of the instructions iload, fload, aload, iload, dload, astore, fstore, astore, lstore, dstore, or ret. The second form applies only to the iinc instruction.

In either case, the wide opcode itself is followed in the compiled code by the opcode of the instruction wide modifies. In either form, two unsigned bytes indexbyte1 and indexbyte2 follow the modified opcode and are assembled into a 16-bit unsigned index to a local variable in the current frame (§3.6), where the value of the index is
wide (cont.)

$(\text{indexbyte1} \ll 8) | \text{indexbyte2}$. The calculated index must be an index into the local variable array of the current frame. Where the wide instruction modifies an $lload$, $dload$, $lstore$, or $dstore$ instruction, the index following the calculated index (index + 1) must also be an index into the local variable array. In the second form, two immediate unsigned bytes $\text{constbyte1}$ and $\text{constbyte2}$ follow $\text{indexbyte1}$ and $\text{indexbyte2}$ in the code stream. Those bytes are also assembled into a signed 16-bit constant, where the constant is $(\text{constbyte1} \ll 8) | \text{constbyte2}$.

The widened bytecode operates as normal, except for the use of the wider index and, in the case of the second form, the larger increment range.

Notes

Although we say that wide “modifies the behavior of another instruction,” the wide instruction effectively treats the bytes constituting the modified instruction as operands, denaturing the embedded instruction in the process. In the case of a modified $iinc$ instruction, one of the logical operands of the $iinc$ is not even at the normal offset from the opcode. The embedded instruction must never be executed directly; its opcode must never be the target of any control transfer instruction.
The Java virtual machine is designed to support the Java programming language. Sun’s JDK software contains both a compiler from source code written in the Java programming language to the instruction set of the Java virtual machine, and a runtime system that implements the Java virtual machine itself. Understanding how one compiler utilizes the Java virtual machine is useful to the prospective compiler writer, as well as to one trying to understand the Java virtual machine itself.

Although this chapter concentrates on compiling source code written in the Java programming language, the Java virtual machine does not assume that the instructions it executes were generated from such code. While there have been a number of efforts aimed at compiling other languages to the Java virtual machine, the current version of the Java virtual machine was not designed to support a wide range of languages. Some languages may be hosted fairly directly by the Java virtual machine. Other languages may be implemented only inefficiently.

Note that the term “compiler” is sometimes used when referring to a translator from the instruction set of a Java virtual machine to the instruction set of a specific CPU. One example of such a translator is a just-in-time (JIT) code generator, which generates platform-specific instructions only after Java virtual machine code has been loaded. This chapter does not address issues associated with code generation, only those associated with compiling source code written in the Java programming language to Java virtual machine instructions.
7.1 Format of Examples

This chapter consists mainly of examples of source code together with annotated listings of the Java virtual machine code that the javac compiler in Sun’s JDK release 1.0.2 generates for the examples. The Java virtual machine code is written in the informal “virtual machine assembly language” output by Sun’s javap utility, distributed with the JDK release. You can use javap to generate additional examples of compiled methods.

The format of the examples should be familiar to anyone who has read assembly code. Each instruction takes the form

\(<index> <opcode> [<operand1> [<operand2>...]][<comment>]\)

The \(<index>\) is the index of the opcode of the instruction in the array that contains the bytes of Java virtual machine code for this method. Alternatively, the \(<index>\) may be thought of as a byte offset from the beginning of the method. The \(<opcode>\) is the mnemonic for the instruction’s opcode, and the zero or more \(<operandN>\) are the operands of the instruction. The optional \(<comment>\) is given in end-of-line comment syntax:

\[8\] bipush 100 // Push int constant 100

Some of the material in the comments is emitted by javap; the rest is supplied by the authors. The \(<index>\) prefacing each instruction may be used as the target of a control transfer instruction. For instance, a goto 8 instruction transfers control to the instruction at index 8. Note that the actual operands of Java virtual machine control transfer instructions are offsets from the addresses of the opcodes of those instructions; these operands are displayed by javap (and are shown in this chapter) as more easily read offsets into their methods.

We preface an operand representing a runtime constant pool index with a hash sign and follow the instruction by a comment identifying the runtime constant pool item referenced, as in

\[10\] ldc #1 // Push float constant 100.0

or

\[9\] invokevirtual #4 // Method Example.addTwo(II)I

For the purposes of this chapter, we do not worry about specifying details such as operand sizes.
7.2 Use of Constants, Local Variables, and Control Constructs

Java virtual machine code exhibits a set of general characteristics imposed by the Java virtual machine’s design and use of types. In the first example we encounter many of these, and we consider them in some detail.

The `spin` method simply spins around an empty for loop 100 times:

```java
void spin() {
    int i;
    for (i = 0; i < 100; i++) {
        ; // Loop body is empty
    }
}
```

A compiler might compile `spin` to

```java
Method void spin()
0  iconst_0 // Push int constant 0
1  istore_1 // Store into local variable 1 (i=0)
2  goto 8 // First time through don’t increment
5  iinc 1 1 // Increment local variable 1 by 1 (i++)
8  iload_1 // Push local variable 1 (i)
9  bipush 100 // Push int constant 100
11 if_icmplt 5 // Compare and loop if less than (i < 100)
14 return // Return void when done
```

The Java virtual machine is stack-oriented, with most operations taking one or more operands from the operand stack of the Java virtual machine’s current frame or pushing results back onto the operand stack. A new frame is created each time a method is invoked, and with it is created a new operand stack and set of local variables for use by that method (see Section 3.6, “Frames”). At any one point of the computation, there are thus likely to be many frames and equally many operand stacks per thread of control, corresponding to many nested method invocations. Only the operand stack in the current frame is active.

The instruction set of the Java virtual machine distinguishes operand types by using distinct bytecodes for operations on its various data types. The method `spin` operates only on values of type `int`. The instructions in its compiled code chosen to operate on typed data (`iconst_0`, `istore_1`, `iinc`, `iload_1`, `if_icmplt`) are all specialized for type `int`. 
The two constants in \texttt{spin}, 0 and 100, are pushed onto the operand stack using two different instructions. The 0 is pushed using an \texttt{iconst\_0} instruction, one of the family of \texttt{iconst\_<i>} instructions. The 100 is pushed using a \texttt{bipush} instruction, which fetches the value it pushes as an immediate operand.

The Java virtual machine frequently takes advantage of the likelihood of certain operands (\texttt{int} constants $-1$, 0, 1, 2, 3, 4 and 5 in the case of the \texttt{iconst\_<i>} instructions) by making those operands implicit in the opcode. Because the \texttt{iconst\_0} instruction knows it is going to push an \texttt{int} 0, \texttt{iconst\_0} does not need to store an operand to tell it what value to push, nor does it need to fetch or decode an operand. Compiling the push of 0 as \texttt{bipush 0} would have been correct, but would have made the compiled code for \texttt{spin} one byte longer. A simple virtual machine would have also spent additional time fetching and decoding the explicit operand each time around the loop. Use of implicit operands makes compiled code more compact and efficient.

The \texttt{int i} in \texttt{spin} is stored as Java virtual machine local variable 1. Because most Java virtual machine instructions operate on values popped from the operand stack rather than directly on local variables, instructions that transfer values between local variables and the operand stack are common in code compiled for the Java virtual machine. These operations also have special support in the instruction set. In \texttt{spin}, values are transferred to and from local variables using the \texttt{istore\_1} and \texttt{iload\_1} instructions, each of which implicitly operates on local variable 1. The \texttt{istore\_1} instruction pops an \texttt{int} from the operand stack and stores it in local variable 1. The \texttt{iload\_1} instruction pushes the value in local variable 1 onto the operand stack.

The use (and reuse) of local variables is the responsibility of the compiler writer. The specialized load and store instructions should encourage the compiler writer to reuse local variables as much as is feasible. The resulting code is faster, more compact, and uses less space in the frame.

Certain very frequent operations on local variables are catered to specially by the Java virtual machine. The \texttt{iinc} instruction increments the contents of a local variable by a one-byte signed value. The \texttt{iinc} instruction in \texttt{spin} increments the first local variable (its first operand) by 1 (its second operand). The \texttt{iinc} instruction is very handy when implementing looping constructs.

The \texttt{for} loop of \texttt{spin} is accomplished mainly by these instructions:
The `bipush` instruction pushes the value 100 onto the operand stack as an `int`, then the `if_icmplt` instruction pops that value off the operand stack and compares it against `i`. If the comparison succeeds (the variable `i` is less than 100), control is transferred to index 5 and the next iteration of the `for` loop begins. Otherwise, control passes to the instruction following the `if_icmplt`.

If the `spin` example had used a data type other than `int` for the loop counter, the compiled code would necessarily change to reflect the different data type. For instance, if instead of an `int` the `spin` example uses a `double`, as shown,

```java
void dspin() {
    double i;
    for (i = 0.0; i < 100.0; i++) {
        ; // Loop body is empty
    }
}
```

the compiled code is

```
Method void dspin()
0  dconst_0 // Push double constant 0.0
1  dstore_1 // Store into local variables 1 and 2
2  goto 9 // First time through don't increment
5  dload_1 // Push local variables 1 and 2
6  dconst_1 // Push double constant 1.0
7  dadd // Add; there is no dinc instruction
8  dstore_1 // Store result in local variables 1 and 2
9  dload_1 // Push local variables 1 and 2
10 ldc2_w #4 // Push double constant 100.0
13 dcmpg // There is no if_icmplt instruction
14 iflt 5 // Compare and loop if less than (i < 100.0)
17 return // Return void when done
```

The instructions that operate on typed data are now specialized for type `double`. (The `ldc2_w` instruction will be discussed later in this chapter.)
Recall that `double` values occupy two local variables, although they are only accessed using the lesser index of the two local variables. This is also the case for values of type `long`. Again for example,

```java
double doubleLocals(double d1, double d2) {
    return d1 + d2;
}
```

becomes

```
Method double doubleLocals(double,double)
0  dload_1 // First argument in local variables 1 and 2
1  dload_3 // Second argument in local variables 3 and 4
2  dadd
3  dreturn
```

Note that local variables of the local variable pairs used to store `double` values in `doubleLocals` must never be manipulated individually.

The Java virtual machine’s opcode size of 1 byte results in its compiled code being very compact. However, 1-byte opcodes also mean that the Java virtual machine instruction set must stay small. As a compromise, the Java virtual machine does not provide equal support for all data types: it is not completely orthogonal (see Table 3.2, “Type support in the Java virtual machine instruction set”).

For example, the comparison of values of type `int` in the `for` statement of `example` `spin` can be implemented using a single `if_icmplt` instruction; however, there is no single instruction in the Java virtual machine instruction set that performs a conditional branch on values of type `double`. Thus, `dspin` must implement its comparison of values of type `double` using a `dcmpg` instruction followed by an `iflt` instruction.

The Java virtual machine provides the most direct support for data of type `int`. This is partly in anticipation of efficient implementations of the Java virtual machine’s operand stacks and local variable arrays. It is also motivated by the frequency of `int` data in typical programs. Other integral types have less direct support. There are no `byte`, `char`, or `short` versions of the `store`, `load`, or `add` instructions, for instance. Here is the `spin` example written using a `short`:
void sspin() {
    short i;
    for (i = 0; i < 100; i++) {
        ; // Loop body is empty
    }
}

It must be compiled for the Java virtual machine, as follows, using instructions operating on another type, most likely int, converting between short and int values as necessary to ensure that the results of operations on short data stay within the appropriate range:

Method void sspin()
0  iconst_0
1  istore_1
2  goto 10
5  iload_1      // The short is treated as though an int
6  iconst_1
7  iadd
8  i2s          // Truncate int to short
9  istore_1
10 iload_1
11 bipush 100
13 if_icmplt 5
16 return

The lack of direct support for byte, char, and short types in the Java virtual machine is not particularly painful, because values of those types are internally promoted to int (byte and short are sign-extended to int, char is zero-extended). Operations on byte, char, and short data can thus be done using int instructions. The only additional cost is that of truncating the values of int operations to valid ranges.

The long and floating-point types have an intermediate level of support in the Java virtual machine, lacking only the full complement of conditional control transfer instructions.
7.3 Arithmetic

The Java virtual machine generally does arithmetic on its operand stack. (The exception is the \textit{iinc} instruction, which directly increments the value of a local variable.) For instance, the \texttt{align2grain} method aligns an \texttt{int} value to a given power of 2:

\begin{verbatim}
int align2grain(int i, int grain) {
    return ((i + grain-1) & ~(grain-1));
}
\end{verbatim}

Operands for arithmetic operations are popped from the operand stack, and the results of operations are pushed back onto the operand stack. Results of arithmetic subcomputations can thus be made available as operands of their nesting computation. For instance, the calculation of $\sim (\text{grain} - 1)$ is handled by these instructions:

\begin{verbatim}
5 iload_2     // Push grain
6 iconst_1    // Push int constant 1
7 isub        // Subtract; push result
8 iconst_m1   // Push int constant -1
9 ixor        // Do XOR; push result
\end{verbatim}

First \texttt{grain} \texttt{} - 1 is calculated using the contents of local variable \texttt{2} and an immediate \texttt{int} value \texttt{1}. These operands are popped from the operand stack and their difference pushed back onto the operand stack. The difference is thus immediately available for use as one operand of the \texttt{ixor} instruction. (Recall that $\sim x = \sim -1^x$.) Similarly, the result of the \texttt{ixor} instruction becomes an operand for the subsequent \texttt{iand} instruction.

The code for the entire method follows:

\begin{verbatim}
Method int align2grain(int, int)
 0 iload_1
 1 iload_2
 2 iadd
 3 icnst_1
 4 isub
 5 iload_2
 6 icnst_1
\end{verbatim}
7  isub
8  iconst_m1
9  ixor
10 iand
11 ireturn

7.4 Accessing the Runtime Constant Pool

Many numeric constants, as well as objects, fields, and methods, are accessed via
the runtime constant pool of the current class. Object access is considered later
(§7.8). Data of types int, long, float, and double, as well as references to
instances of class String, are managed using the ldc, ldc_w, and ldc2_w instruc-
tions.

The ldc and ldc_w instructions are used to access values in the runtime con-
stant pool (including instances of class String) of types other than double and
long. The ldc_w instruction is used in place of ldc only when there is a large num-
ber of runtime constant pool items and a larger index is needed to access an item.
The ldc2_w instruction is used to access all values of types double and long; there is no non-wide variant.

Integral constants of types byte, char, or short, as well as small int values,
may be compiled using the bipush, sipush, or iconst_<i> instructions, as seen ear-
lier (§7.2). Certain small floating-point constants may be compiled using the
fconst_<f> and dconst_<d> instructions.

In all of these cases, compilation is straightforward. For instance, the con-
stants for

```java
void useManyNumeric() {
    int i = 100;
    int j = 1000000;
    long l1 = 1;
    long l2 = 0xffffffff;
    double d = 2.2;
    ...do some calculations...
}
```

are set up as follows:
Method void useManyNumeric()
0 bipush 100 // Push a small int with bipush
2 istore_1
3 ldc #1 // Push int constant 1000000; a larger int
// value uses ldc
5 istore_2
6 lconst_1 // A tiny long value uses short, fast lconst_1
7 lstore_3
8 ldc2_w #6 // Push long 0xffffffff (that is, an int −1); any
// long constant value can be pushed using ldc2_w
11 lstore_5
13 ldc2_w #8 // Push double constant 2.200000; uncommon
// double values are also pushed using ldc2_w
16 dstore 7
...do those calculations...

7.5 More Control Examples

Compilation of for statements was shown in an earlier section (§7.2). Most of the Java
programming language’s other control constructs (if-then-else, do, while, break, and continue) are also compiled in the obvious ways. The compilation of
switch statements is handled in a separate section (Section 7.10, “Compiling
Switches”), as are the compilation of exceptions (Section 7.12, “Throwing and Han-
dling Exceptions”) and the compilation of finally clauses (Section 7.13, “Compiling
finally”).

As a further example, a while loop is compiled in an obvious way, although
the specific control transfer instructions made available by the Java virtual
machine vary by data type. As usual, there is more support for data of type int, for example:

```java
void whileInt() {
    int i = 0;
    while (i < 100) {
        i++;
    }
}
```

is compiled to
Note that the test of the `while` statement (implemented using the `if_icmplt` instruction) is at the bottom of the Java virtual machine code for the loop. (This was also the case in the spin examples earlier.) The test being at the bottom of the loop forces the use of a `goto` instruction to get to the test prior to the first iteration of the loop. If that test fails, and the loop body is never entered, this extra instruction is wasted. However, `while` loops are typically used when their body is expected to be run, often for many iterations. For subsequent iterations, putting the test at the bottom of the loop saves a Java virtual machine instruction each time around the loop: if the test were at the top of the loop, the loop body would need a trailing `goto` instruction to get back to the top.

Control constructs involving other data types are compiled in similar ways, but must use the instructions available for those data types. This leads to somewhat less efficient code because more Java virtual machine instructions are needed, for example:

```java
void whileDouble() {
    double i = 0.0;
    while (i < 100.1) {
        i++;
    }
}
```

is compiled to

```java
Method void whileDouble()
0  dconst_0
1  dstore_1
2  goto 9
5  dload_1
```
Each floating-point type has two comparison instructions: \texttt{fcmpl} and \texttt{fcmpg} for type \texttt{float}, and \texttt{dcmpl} and \texttt{dcmpg} for type \texttt{double}. The variants differ only in their treatment of \texttt{NaN}. \texttt{NaN} is unordered, so all floating-point comparisons fail if either of their operands is \texttt{NaN}. The compiler chooses the variant of the comparison instruction for the appropriate type that produces the same result whether the comparison fails on non-\texttt{NaN} values or encounters a \texttt{NaN}.

For instance:

```java
int lessThan100(double d) {
  if (d < 100.0) {
    return 1;
  } else {
    return -1;
  }
}
```

compiles to

```
Method int lessThan100(double)
  0 dload_1
  1 ldc2_w #4 // Push double constant 100.0
  4 dcmpg   // Push 1 if d is NaN or d > 100.0;
             // push 0 if d == 100.0
  5 ifge 10 // Branch on 0 or 1
  8 iconst_1
  9 ireturn
10 iconst_m1
11 ireturn
```
If \( d \) is not NaN and is less than 100.0, the dcmpg instruction pushes an int \(-1\) onto the operand stack, and the ifge instruction does not branch. Whether \( d \) is greater than 100.0 or is NaN, the dcmpg instruction pushes an int \( 1 \) onto the operand stack, and the ifge branches. If \( d \) is equal to 100.0, the dcmpg instruction pushes an int \( 0 \) onto the operand stack, and the ifge branches.

The dcmpl instruction achieves the same effect if the comparison is reversed:

```java
int greaterThan100(double d) {
    if (d > 100.0) {
        return 1;
    } else {
        return -1;
    }
}
```

becomes

```
Method int greaterThan100(double)
    0  dload_1
    1  ldc2_w #4 // Push double constant 100.0
    4  dcmpl // Push -1 if d is NaN or d < 100.0;
           // push 0 if d == 100.0
    5  ifle 10 // Branch on 0 or -1
    8  iconst_1
    9  ireturn
   10  iconst_m1
   11  ireturn
```

Once again, whether the comparison fails on a non-NaN value or because it is passed a NaN, the dcmpl instruction pushes an int value onto the operand stack that causes the ifle to branch. If both of the dcmp instructions did not exist, one of the example methods would have had to do more work to detect NaN.
7.6 Receiving Arguments

If \( n \) arguments are passed to an instance method, they are received, by convention, in the local variables numbered 1 through \( n \) of the frame created for the new method invocation. The arguments are received in the order they were passed. For example:

\[
\text{int addTwo(int i, int j) \{ \\
\quad \text{return } i + j; \\
\}}
\]

compiles to

\[
\text{Method int addTwo(int,int)} \\
0 \quad \text{iload}_1 \quad \text{// Push value of local variable 1 (i)} \\
1 \quad \text{iload}_2 \quad \text{// Push value of local variable 2 (j)} \\
2 \quad \text{iadd} \quad \text{// Add; leave int result on operand stack} \\
3 \quad \text{ireturn} \quad \text{// Return int result}
\]

By convention, an instance method is passed a reference to its instance in local variable 0. In the Java programming language the instance is accessible via the this keyword.

Class (static) methods do not have an instance, so for them this use of local variable zero is unnecessary. A class method starts using local variables at index zero. If the addTwo method were a class method, its arguments would be passed in a similar way to the first version:

\[
\text{static int addTwoStatic(int i, int j) \{ \\
\quad \text{return } i + j; \\
\}}
\]

compiles to

\[
\text{Method int addTwoStatic(int,int)} \\
0 \quad \text{iload}_0 \\
1 \quad \text{iload}_1 \\
2 \quad \text{iadd} \\
3 \quad \text{ireturn}
\]

The only difference is that the method arguments appear starting in local variable 0 rather than 1.
7.7 Invoking Methods

The normal method invocation for an instance method dispatches on the runtime type of the object. (They are virtual, in C++ terms.) Such an invocation is implemented using the `invokevirtual` instruction, which takes as its argument an index to a runtime constant pool entry giving the internal form of the binary name of the class type of the object, the name of the method to invoke, and that method’s descriptor (§4.3.3). To invoke the `addTwo` method, defined earlier as an instance method, we might write

```java
int add12and13() {
    return addTwo(12, 13);
}
```

This compiles to

```
Method int add12and13()
  0  aload_0       // Push local variable 0 (this)
  1  bipush 12    // Push int constant 12
  3  bipush 13    // Push int constant 13
  5  invokevirtual #4 // Method Example.addtwo(II)I
  8  ireturn       // Return int on top of operand stack; it is
                    // the int result of addTwo()
```

The invocation is set up by first pushing a reference to the current instance, `this`, onto the operand stack. The method invocation’s arguments, `int` values 12 and 13, are then pushed. When the frame for the `addTwo` method is created, the arguments passed to the method become the initial values of the new frame’s local variables. That is, the reference for `this` and the two arguments, pushed onto the operand stack by the invoker, will become the initial values of local variables 0, 1, and 2 of the invoked method.

Finally, `addTwo` is invoked. When it returns, its `int` return value is pushed onto the operand stack of the frame of the invoker, the `add12and13` method. The return value is thus put in place to be immediately returned to the invoker of `add12and13`.

The return from `add12and13` is handled by the `ireturn` instruction of `add12and13`. The `ireturn` instruction takes the `int` value returned by `addTwo`, on the operand stack of the current frame, and pushes it onto the operand stack of the frame of the invoker. It then returns control to the invoker, making the invoker’s frame current. The Java virtual machine provides distinct return instructions for
many of its numeric and reference data types, as well as a return instruction for methods with no return value. The same set of return instructions is used for all varieties of method invocations.

The operand of the invokevirtual instruction (in the example, the runtime constant pool index #4) is not the offset of the method in the class instance. The compiler does not know the internal layout of a class instance. Instead, it generates symbolic references to the methods of an instance, which are stored in the runtime constant pool. Those runtime constant pool items are resolved at runtime to determine the actual method location. The same is true for all other Java virtual machine instructions that access class instances.

Invoking addTwoStatic, a class (static) variant of addTwo, is similar, as shown:

```java
int add12and13() {
    return addTwoStatic(12, 13);
}
```

although a different Java virtual machine method invocation instruction is used:

```
Method int add12and13()
 0  bipush 12
 2  bipush 13
 4  invokestatic #3       // Method Example.addTwoStatic(II)I
 7  ireturn
```

Compiling an invocation of a class (static) method is very much like compiling an invocation of an instance method, except this is not passed by the invoker. The method arguments will thus be received beginning with local variable 0 (see Section 7.6, “Receiving Arguments”). The invokevirtual instruction is always used to invoke class methods.

The invokespecial instruction must be used to invoke instance initialization methods (see Section 7.8, “Working with Class Instances”). It is also used when invoking methods in the superclass (super) and when invoking private methods. For instance, given classes Near and Far declared as

```java
class Near {
    int it;
    public int getItNear() {
        return getIt();
    }
}
private int getIt() {
    return it;
}

class Far extends Near {
    int getItFar() {
        return super.getItNear();
    }
}

the method Near.getItNear (which invokes a private method) becomes

Method int getItNear()
0  aload_0
1  invokespecial #5 // Method Near.getIt()I
4  ireturn

The method Far.getItFar (which invokes a superclass method) becomes

Method int getItFar()
0  aload_0
1  invokespecial #4 // Method Near.getItNear()I
4  ireturn

Note that methods called using the invokespecial instruction always pass this to the invoked method as its first argument. As usual, it is received in local variable 0.

7.8 Working with Class Instances

Java virtual machine class instances are created using the Java virtual machine’s new instruction. Recall that at the level of the Java virtual machine, a constructor appears as a method with the compiler-supplied name <init>. This specially named method is known as the instance initialization method (§3.9). Multiple instance initialization methods, corresponding to multiple constructors, may exist for a given class. Once the class instance has been created and its instance variables, including those of the class and all of its superclasses, have been initialized to their default values, an instance initialization method of the new class instance is invoked. For example:
Object create() {
    return new Object();
}

compiles to

Method java.lang.Object create()
  0  new #1      // Class java.lang.Object
  3  dup
  4  invokespecial #4 // Method java.lang.Object.<init>()V
  7  areturn

Class instances are passed and returned (as reference types) very much like numeric values, although type reference has its own complement of instructions, for example:

    int i;          // An instance variable
    MyObj example() {
        MyObj o = new MyObj();
        return silly(o);
    }
    MyObj silly(MyObj o) {
        if (o != null) {
            return o;
        } else {
            return o;
        }
    }

becomes
Method MyObj example()
0    new #2    // Class MyObj
3    dup
4    invokespecial #5    // Method MyObj.<init>()V
7    astore_1
8    aload_0
9    aload_1
10   invokevirtual #4
     // Method Example.silly(LMyObj;LMyObj;
13   areturn

Method MyObj silly(MyObj)
0    aload_1
1    ifnull 6
4    aload_1
5    areturn
6    aload_1
7    areturn

The fields of a class instance (instance variables) are accessed using the `get-field` and `putfield` instructions. If \( i \) is an instance variable of type \( \text{int} \), the methods `setIt` and `getIt`, defined as

```java
void setIt(int value) {
    i = value;
}

int getIt() {
    return i;
}
```

become

Method void setIt(int)
0    aload_0
1    iload_1
2    putfield #4    // Field Example.i I
5    return
Method int getIt()  
0  aload_0  
1  getfield #4 // Field Example.i  
4  ireturn

As with the operands of method invocation instructions, the operands of the putfield and getfield instructions (the runtime constant pool index #4) are not the offsets of the fields in the class instance. The compiler generates symbolic references to the fields of an instance, which are stored in the runtime constant pool. Those runtime constant pool items are resolved at runtime to determine the location of the field within the referenced object.

### 7.9 Arrays

Java virtual machine arrays are also objects. Arrays are created and manipulated using a distinct set of instructions. The newarray instruction is used to create an array of a numeric type. The code

```java
void createBuffer() {
    int buffer[];
    int bufsz = 100;
    int value = 12;
    buffer = new int[bufsz];
    buffer[10] = value;
    value = buffer[11];
}
```

might be compiled to
Method void createBuffer()
0 bipush 100 // Push int constant 100 (buksz)
2 istore_2 // Store bufsz in local variable 2
3 bipush 12 // Push int constant 12 (value)
5 istore_3 // Store value in local variable 3
6 iload_2 // Push bufsz...
7 anewarray int // ...and create new array of int of that length
9 astore_1 // Store new array in buffer
10 aload_1 // Push buffer
11 bipush 10 // Push int constant 10
13 iload_3 // Push value
14 iastore // Store value at buffer[10]
15 aload_1 // Push buffer
16 bipush 11 // Push int constant 11
18 iaload // Push value at buffer[11]...
19 istore_3 // ...and store it in value
20 return

The anewarray instruction is used to create a one-dimensional array of object references, for example:

void createThreadArray() {
    Thread threads[];
    int count = 10;
    threads = new Thread[count];
    threads[0] = new Thread();
}

becomes

Method void createThreadArray()
0 bipush 10 // Push int constant 10
2 istore_2 // Initialize count to that
3 iload_2 // Push count, used by anewarray
4 anewarray class #1 // Create new array of class Thread
7 astore_1 // Store new array in threads
8 aload_1 // Push value of threads
9 icast_0 // Push int constant 0
10 anew #1 // Create instance of class Thread
COMPILING FOR THE JAVA VIRTUAL MACHINE

13 dup // Make duplicate reference...
14 invokespecial #5 // ...to pass to instance initialization method
               // Method java.lang.Thread.<init>()V
17 aastore // Store new Thread in array at 0
18 return

The **anewarray** instruction can also be used to create the first dimension of a multi-dimensional array. Alternatively, the **multianewarray** instruction can be used to create several dimensions at once. For example, the three-dimensional array:

\[
\text{int}[][][] \text{create3DArray}() \{
\text{int grid}[][][];
\text{grid} = \text{new int}[10][5][];
\text{return grid;}
\}
\]

is created by

\[\text{Method } \text{int create3DArray}()[][][]\]

\[
0 \text{ bipush 10} \quad \text{// Push int 10 (dimension one)}
2 \text{ icanst_5} \quad \text{// Push int 5 (dimension two)}
3 \text{ multianewarray #1 dim #2} \quad \text{// Class }[][][], \text{a three}
\quad \text{// dimensional int array;}
\quad \text{// only create first two}
\quad \text{// dimensions}
7 \text{ astore_1} \quad \text{// Store new array...}
8 \text{ aload_1} \quad \text{// ...then prepare to return it}
9 \text{ areturn}
\]

The first operand of the **multianewarray** instruction is the runtime constant pool index to the array class type to be created. The second is the number of dimensions of that array type to actually create. The **multianewarray** instruction can be used to create all the dimensions of the type, as the code for **create3DArray** shows. Note that the multidimensional array is just an object and so is loaded and returned by an **aload_1** and **areturn** instruction, respectively. For information about array class names, see Section 4.4.1.

All arrays have associated lengths, which are accessed via the **arraylength** instruction.
7.10 Compiling Switches

Compilation of switch statements uses the tableswitch and lookupswitch instructions. The tableswitch instruction is used when the cases of the switch can be efficiently represented as indices into a table of target offsets. The default target of the switch is used if the value of the expression of the switch falls outside the range of valid indices. For instance,

```java
int chooseNear(int i) {
    switch (i) {
    case 0: return 0;
    case 1: return 1;
    case 2: return 2;
    default: return -1;
    }
}
```

compiles to

```
Method int chooseNear(int)
0  iload_1  // Push local variable 1 (argument i)
1  tableswitch 0 to 2:  // Valid indices are 0 through 2
    0: 28  // If i is 0, continue at 28
    1: 30  // If i is 1, continue at 30
    2: 32  // If i is 2, continue at 32
    default:34  // Otherwise, continue at 34
28  iconst_0  // i was 0; push int constant 0...
29  ireturn  // ...and return it
30  iconst_1  // i was 1; push int constant 1...
31  ireturn  // ...and return it
32  iconst_2  // i was 2; push int constant 2...
33  ireturn  // ...and return it
34  iconst_m1  // otherwise push int constant -1...
35  ireturn  // ...and return it
```

The Java virtual machine’s tableswitch and lookupswitch instructions operate only on int data. Because operations on byte, char, or short values are internally promoted to int, a switch whose expression evaluates to one of those types is compiled as though it evaluated to type int. If the chooseNear
method had been written using type `short`, the same Java virtual machine instructions would have been generated as when using type `int`. Other numeric types must be narrowed to type `int` for use in a `switch`.

Where the cases of the `switch` are sparse, the table representation of the `tableswitch` instruction becomes inefficient in terms of space. The `lookupswitch` instruction may be used instead. The `lookupswitch` instruction pairs `int` keys (the values of the case labels) with target offsets in a table. When a `lookupswitch` instruction is executed, the value of the expression of the `switch` is compared against the keys in the table. If one of the keys matches the value of the expression, execution continues at the associated target offset. If no key matches, execution continues at the `default` target. For instance, the compiled code for

```java
int chooseFar(int i) {
    switch (i) {
        case -100: return -1;
        case 0: return 0;
        case 100: return 1;
        default: return -1;
    }
}
```

looks just like the code for `chooseNear`, except for the `lookupswitch` instruction:

```java
Method int chooseFar(int)
0  iload_1
1  lookupswitch 3:  
   -100: 36
   0: 38
   100: 40
   default:42
36  iconst_m1
37  ireturn
38  iconst_0
39  ireturn
40  iconst_1
41  ireturn
42  iconst_m1
43  ireturn
```
The Java virtual machine specifies that the table of the \textit{lookupswitch} instruction must be sorted by key so that implementations may use searches more efficient than a linear scan. Even so, the \textit{lookupswitch} instruction must search its keys for a match rather than simply perform a bounds check and index into a table like \textit{tableswitch}. Thus, a \textit{tableswitch} instruction is probably more efficient than a \textit{lookupswitch} where space considerations permit a choice.

### 7.11 Operations on the Operand Stack

The Java virtual machine has a large complement of instructions that manipulate the contents of the operand stack as untyped values. These are useful because of the Java virtual machine’s reliance on deft manipulation of its operand stack. For instance,

```java
public long nextIndex() {
    return index++;}
```

```java
private long index = 0;
```

is compiled to

```java
Method long nextIndex()
0  aload_0  // Push this
1   dup    // Make a copy of it
2  getfield #4  // One of the copies of this is consumed
   // pushing long field index,
   // above the original this
5  dup2_x1  // The long on top of the operand stack is
   // inserted into the operand stack below the
   // original this
6  lconst_1  // Push long constant 1
7  ladd    // The index value is incremented...
8  putfield #4  // ...and the result stored back in the field
11 lreturn  // The original value of index is left on
   // top of the operand stack, ready to be returned
```

Note that the Java virtual machine never allows its operand stack manipulation instructions to modify or break up individual values on the operand stack.
7.12 Throwing and Handling Exceptions

Exceptions are thrown from programs using the throw keyword. Its compilation is simple:

```java
void cantBeZero(int i) throws TestExc {
    if (i == 0) {
        throw new TestExc();
    }
}
```

becomes

```
Method void cantBeZero(int)
0 iload_1 // Push argument 1 (i)
1 ifne 12 // If i==0, allocate instance and throw
4 new #1 // Create instance of TestExc
7 dup // One reference goes to the constructor
8 invokespecial #7 // Method TestExc.<init>()V
11 athrow // Second reference is thrown
12 return // Never get here if we threw TestExc
```

Compilation of try-catch constructs is straightforward. For example,

```java
void catchOne() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
    }
}
```

is compiled as

```
Method void catchOne()
0 aload_0 // Beginning of try block
1 invokevirtual #6 // Method Example.tryItOut()V
4 return // End of try block; normal return
5 astore_1 // Store thrown value in local variable 1
6 aload_0 // Push this
```
7  aload_1  // Push thrown value
8  invokevirtual #5  // Invoke handler method:
     // Example.handleExc(LTestExc;)V
11  return  // Return after handling TestExc

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>5</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TestExc</td>
</tr>
</tbody>
</table>

Looking more closely, the try block is compiled just as it would be if the try were not present:

```java
Method  void catchOne()
0  aload_0  // Beginning of try block
1  invokevirtual #4  // Method Example.tryItOut()V
4  return  // End of try block; normal return

If no exception is thrown during the execution of the try block, it behaves as though the try were not there: tryItOut is invoked and catchOne returns.
```

Following the try block is the Java virtual machine code that implements the single catch clause:

```java
5  astore_1  // Store thrown value in local variable 1
6  aload_0  // Push this
7  aload_1  // Push thrown value
8  invokevirtual #5  // Invoke handler method:
     // Example.handleExc(LTestExc;)V
11  return  // Return after handling TestExc

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>5</td>
<td>Class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TestExc</td>
</tr>
</tbody>
</table>
```

The invocation of handleExc, the contents of the catch clause, is also compiled like a normal method invocation. However, the presence of a catch clause causes the compiler to generate an exception table entry (§3.10, §4.7.3). The exception table for the catchOne method has one entry corresponding to the one argument (an instance of class TestExc) that the catch clause of catchOne can handle. If some value that is an instance of TestExc is thrown during execution of the instructions between in-dices 0 and 4 in catchOne, control is transferred to the Java virtual machine code at index 5, which implements the block of the catch clause. If the
value that is thrown is not an instance of TestExc, the catch clause of catchOne cannot handle it. Instead, the value is rethrown to the invoker of catchOne.

A try may have multiple catch clauses:

```java
void catchTwo() {
    try {
        tryItOut();
    } catch (TestExc1 e) {
        handleExc(e);
    } catch (TestExc2 e) {
        handleExc(e);
    }
}
```

Multiple catch clauses of a given try statement are compiled by simply appending the Java virtual machine code for each catch clause one after the other and adding entries to the exception table, as shown:

```
Method void catchTwo()
  0  aload_0  // Begin try block
  1  invokevirtual #5 // Method Example.tryItOut()V
  4  return  // End of try block; normal return
  5  astore_1  // Beginning of handler for TestExc1;
             // Store thrown value in local variable 1
  6  aload_0  // Push this
  7  aload_1  // Push thrown value
  8  invokevirtual #7 // Invoke handler method:
             // Example.handleExc(LTestExc1;)V
 11  return  // Return after handling TestExc1
 12  astore_1  // Beginning of handler for TestExc2;
             // Store thrown value in local variable 1
 13  aload_0  // Push this
 14  aload_1  // Push thrown value
 15  invokevirtual #7 // Invoke handler method:
             // Example.handleExc(LTestExc2;)V
 18  return  // Return after handling TestExc2
```

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
</table>

If during the execution of the try clause (between indices 0 and 4) a value is thrown that matches the parameter of one or more of the catch clauses (the value is an instance of one or more of the parameters), the first (innermost) such catch clause is selected. Control is transferred to the Java virtual machine code for the block of that catch clause. If the value thrown does not match the parameter of any of the catch clauses of catchTwo, the Java virtual machine rethrows the value without invoking code in any catch clause of catchTwo.

Nested try-catch statements are compiled very much like a try statement with multiple catch clauses:

```java
void nestedCatch() {
    try {
        try {
            tryItOut();
        } catch (TestExc1 e) {
            handleExc1(e);
        }
        } catch (TestExc2 e) {
            handleExc2(e);
        }
    }
}
```

becomes

```
Method void nestedCatch()
0  aload_0  // Begin try block
1  invokevirtual #8 // Method Example.tryItOut()V
4  return  // End of try block; normal return
5  astore_1 // Beginning of handler for TestExc1;
       // Store thrown value in local variable 1
6  aload_0 // Push this
7  astore_1 // Push thrown value
8  invokevirtual #7 // Invoke handler method:
       // Example.handleExc1(LTestExc1;)V
11 return  // Return after handling TestExc1
12 astore_1 // Beginning of handler for TestExc2;
```
// Store thrown value in local variable 1
13  aload_0   // Push this
14  aload_1   // Push thrown value
15  invokevirtual #6 // Invoke handler method:
      // Example.handleExc2(LTestExc2;)V
18  return    // Return after handling TestExc2

Exception table:
<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>5</td>
<td>Class TestExc1</td>
</tr>
<tr>
<td>0</td>
<td>12</td>
<td>12</td>
<td>Class TestExc2</td>
</tr>
</tbody>
</table>

The nesting of catch clauses is represented only in the exception table. The Java virtual machine does not enforce nesting of or any ordering of the exception table entries (§3.10). However, because try-catch constructs are structured, a compiler can always order the entries of the exception handler table such that, for any thrown exception and any program counter value in that method, the first exception handler that matches the thrown exception corresponds to the innermost matching catch clause.

For instance, if the invocation of tryItOut (at index 1) threw an instance of TestExc1, it would be handled by the catch clause that invokes handleExc1. This is so even though the exception occurs within the bounds of the outer catch clause (catching TestExc2) and even though that outer catch clause might otherwise have been able to handle the thrown value.

As a subtle point, note that the range of a catch clause is inclusive on the “from” end and exclusive on the “to” end (§4.7.3). Thus, the exception table entry for the catch clause catching TestExc1 does not cover the return instruction at offset 4. However, the exception table entry for the catch clause catching TestExc2 does cover the return instruction at offset 11. Return instructions within nested catch clauses are included in the range of instructions covered by nesting catch clauses.
7.13 Compiling finally

Compilation of a try-finally statement is similar to that of try-catch. Prior to transferring control outside the try statement, whether that transfer is normal or abrupt, because an exception has been thrown, the finally clause must first be executed. For this simple example

```java
void tryFinally() {
    try {
        tryItOut();
    } finally {
        wrapItUp();
    }
}
```

the compiled code is

```
Method void tryFinally()
0   aload_0     // Beginning of try block
1   invokevirtual #6 // Method Example.tryItOut()V
4   jsr 14      // Call finally block
7   return     // End of try block
8   astore_1    // Beginning of handler for any throw
9   jsr 14      // Call finally block
12  aload_1     // Push thrown value
13  athrow      // ...and rethrow the value to the invoker
14  astore_2    // Beginning of finally block
15  aload_0     // Push this
16  invokevirtual #5 // Method Example.wrapItUp()V
19  ret 2       // Return from finally block
```

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>any</td>
</tr>
</tbody>
</table>

There are four ways for control to pass outside of the try statement: by falling through the bottom of that block, by returning, by executing a break or continue statement, or by raising an exception. If tryItOut returns without raising an exception, control is transferred to the finally block using a jsr instruction. The jsr 14 instruction at index 4 makes a “subroutine call” to the code for the finally block at
index 14 (the finally block is compiled as an embedded subroutine). When the finally block completes, the ret 2 instruction returns control to the instruction following the jsr instruction at index 4.

In more detail, the subroutine call works as follows: The jsr instruction pushes the address of the following instruction (return at index 7) onto the operand stack before jumping. The astore_2 instruction that is the jump target stores the address on the operand stack into local variable 2. The code for the finally block (in this case the aload_0 and invokevirtual instructions) is run. Assuming execution of that code completes normally, the ret instruction retrieves the address from local variable 2 and resumes execution at that address. The return instruction is executed, and tryFinally returns normally.

A try statement with a finally clause is compiled to have a special exception handler, one that can handle any exception thrown within the try statement. If tryItOut throws an exception, the exception table for tryFinally is searched for an appropriate exception handler. The special handler is found, causing execution to continue at index 8. The astore_1 instruction at index 8 stores the thrown value into local variable 1. The following jsr instruction does a subroutine call to the code for the finally block. Assuming that code returns normally, the aload_1 instruction at index 12 pushes the thrown value back onto the operand stack, and the following athrow instruction rethrows the value.

Compiling a try statement with both a catch clause and a finally clause is more complex:

```java
void tryCatchFinally() {
    try {
        tryItOut();
    } catch (TestExc e) {
        handleExc(e);
    } finally {
        wrapItUp();
    }
}
```

becomes

```
Method void tryCatchFinally()
 0  aload_0 // Beginning of try block
 1  invokevirtual #4 // Method Example.tryItOut()V
 4  goto 16 // Jump to finally block
```
If the try statement completes normally, the goto instruction at index 4 jumps to the subroutine call for the finally block at index 16. The finally block at index 26 is executed, control returns to the return instruction at index 19, and tryCatchFinally returns normally.

If tryItOut throws an instance of TestExc, the first (innermost) applicable exception handler in the exception table is chosen to handle the exception. The code for that exception handler, beginning at index 7, passes the thrown value to handleExc and on its return makes the same subroutine call to the finally block at index 26 as in the normal case. If an exception is not thrown by handleExc, tryCatchFinally returns normally.

If tryItOut throws a value that is not an instance of TestExc or if handleExc itself throws an exception, the condition is handled by the second entry in the

---

1 This goto instruction is strictly unnecessary, but is generated by the javac compiler of Sun’s JDK release 1.0.2.
exception table, which handles any value thrown between indices 0 and 16. That exception handler transfers control to index 20, where the thrown value is first stored in local variable 1. The code for the finally block at index 26 is called as a subroutine. If it returns, the thrown value is retrieved from local variable 1 and rethrown using the athrow instruction. If a new value is thrown during execution of the finally clause, the finally clause aborts, and tryCatchFinally returns abruptly, throwing the new value to its invoker.

## 7.14 Annotations

The representation of annotations in class files is described in §4.7.16 and §4.7.17, which make it clear how to represent annotations on types, fields and methods in the class file format. Package annotations require additional rules, given here.

When a compiler encounters an annotated package declaration that must be made available at runtime, it must emit a class file that represents an interface whose name is the internal form (§4.2.1) of package-name.package-info. The interface has the default access level (“package-private”) and no superinterfaces. The ACC_INTERFACE and ACC_ABSTRACT flags (Table 4.1) of the ClassFile structure (§4.1) must be set. If the emitted class file version number is less than 50.0, then the ACC_SYNTHETIC flag must be unset; if the class file version number is 50.0 or above, then the ACC_SYNTHETIC flag must be set. The only elements declared by the interface must be those implied by The Java Language Specification, Third Edition (JLS3 §9.2).

The package-level annotations are stored in the RuntimeVisibleAnnotations (§4.7.16) and RuntimeInvisibleAnnotations (§4.7.17) attributes of the ClassFile structure (§4.1) of this interface.

## 7.15 Synchronization

Synchronization in the Java virtual machine is implemented by monitor entry and exit, either explicitly (by use of the monitorenter and monitorexit instructions) or implicitly (by the method invocation and return instructions).

For code written in the Java programming language, perhaps the most common form of synchronization is the synchronized method. A synchronized method is not normally implemented using monitorenter and monitorexit. Rather, it is simply distinguished in the runtime constant pool by the ACC_SYNCHRONIZED flag.
flag, which is checked by the method invocation instructions (see Section 3.11.10, “Synchronization”).

The monitorenter and monitorexit instructions enable the compilation of synchronized statements. For example:

```java
void onlyMe(Foo f) {
    synchronized(f) {
        doSomething();
    }
}
```

is compiled to

```java
Method  void onlyMe(Foo)
0   aload_1        // Push f
1   astore_2       // Store it in local variable 2
2   aload_2        // Push local variable 2 (f)
3   monitorenter    // Enter the monitor associated with f
4   aload_0        // Holding the monitor, pass this and...
5   invokevirtual #5 // ...call Example.doSomething()V
8   aload_2        // Push local variable 2 (f)
9   monitorexit     // Exit the monitor associated with f
10  return         // Return normally
11  aload_2        // In case of any throw, end up here
12  monitorexit     // Be sure to exit monitor...
13  athrow         // ...then rethrow the value to the invoker

Exception table:
 FromTo  Target  Type
  4   9     11   any
```

The compiler ensures that at any method invocation completion, a monitorexit instruction will have been executed for each monitorenter instruction executed since the method invocation. This is the case whether the method invocation completes normally (§3.6.4) or abruptly (§3.6.5). To enforce proper pairing of monitorenter and monitorexit instructions on abrupt method invocation completion, the compiler generates exception handlers (§3.10) that will match any exception and whose associated code executes the necessary monitorexit instructions.
In *The Java™ Virtual Machine Specification, Second Edition*, Chapter 8 detailed the low-level actions that explained the interaction of Java virtual machine threads with a shared main memory. It was adapted from Chapter 17 of *The Java™ Language Specification, First Edition*.

Chapter 17 in *The Java™ Language Specification, Third Edition*, was updated to incorporate *The Java™ Memory Model and Thread Specification* produced by the JSR-133 Expert Group. The reader is referred to that chapter for information about threads and locks.
**CHAPTER 9**

**Opcode Mnemonics by Opcode**

This chapter gives the mapping from Java virtual machine instruction opcodes, including the reserved opcodes (§6.2), to the mnemonics for the instructions represented by those opcodes.

<table>
<thead>
<tr>
<th>Opcode (Hex)</th>
<th>Instruction</th>
<th>Opcode (Hex)</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (0x00)</td>
<td><code>nop</code></td>
<td>24 (0x18)</td>
<td><code>dload</code></td>
</tr>
<tr>
<td>1 (0x01)</td>
<td><code>aconst_null</code></td>
<td>25 (0x19)</td>
<td><code>aload</code></td>
</tr>
<tr>
<td>2 (0x02)</td>
<td><code>iconst_m1</code></td>
<td>26 (0x1a)</td>
<td><code>iload_0</code></td>
</tr>
<tr>
<td>3 (0x03)</td>
<td><code>iconst_0</code></td>
<td>27 (0x1b)</td>
<td><code>iload_1</code></td>
</tr>
<tr>
<td>4 (0x04)</td>
<td><code>iconst_1</code></td>
<td>28 (0x1c)</td>
<td><code>iload_2</code></td>
</tr>
<tr>
<td>5 (0x05)</td>
<td><code>iconst_2</code></td>
<td>29 (0x1d)</td>
<td><code>iload_3</code></td>
</tr>
<tr>
<td>6 (0x06)</td>
<td><code>iconst_3</code></td>
<td>30 (0x1e)</td>
<td><code>lload_0</code></td>
</tr>
<tr>
<td>7 (0x07)</td>
<td><code>iconst_4</code></td>
<td>31 (0x1f)</td>
<td><code>lload_1</code></td>
</tr>
<tr>
<td>8 (0x08)</td>
<td><code>iconst_5</code></td>
<td>32 (0x20)</td>
<td><code>lload_2</code></td>
</tr>
<tr>
<td>9 (0x09)</td>
<td><code>lconst_0</code></td>
<td>33 (0x21)</td>
<td><code>lload_3</code></td>
</tr>
<tr>
<td>10 (0x0a)</td>
<td><code>lconst_1</code></td>
<td>34 (0x22)</td>
<td><code>fload_0</code></td>
</tr>
<tr>
<td>11 (0x0b)</td>
<td><code>fconst_0</code></td>
<td>35 (0x23)</td>
<td><code>fload_1</code></td>
</tr>
<tr>
<td>12 (0x0c)</td>
<td><code>fconst_1</code></td>
<td>36 (0x24)</td>
<td><code>fload_2</code></td>
</tr>
<tr>
<td>13 (0x0d)</td>
<td><code>fconst_2</code></td>
<td>37 (0x25)</td>
<td><code>fload_3</code></td>
</tr>
<tr>
<td>14 (0x0e)</td>
<td><code>fconst_3</code></td>
<td>38 (0x26)</td>
<td><code>dload_0</code></td>
</tr>
<tr>
<td>15 (0x0f)</td>
<td><code>dconst_0</code></td>
<td>39 (0x27)</td>
<td><code>dload_1</code></td>
</tr>
<tr>
<td>16 (0x10)</td>
<td><code>bipush</code></td>
<td>40 (0x28)</td>
<td><code>dload_2</code></td>
</tr>
<tr>
<td>17 (0x11)</td>
<td><code>sipush</code></td>
<td>41 (0x29)</td>
<td><code>dload_3</code></td>
</tr>
<tr>
<td>18 (0x12)</td>
<td><code>ldc</code></td>
<td>42 (0x2a)</td>
<td><code>aload_0</code></td>
</tr>
<tr>
<td>19 (0x13)</td>
<td><code>ldc_w</code></td>
<td>43 (0x2b)</td>
<td><code>aload_1</code></td>
</tr>
<tr>
<td>20 (0x14)</td>
<td><code>ldc2_w</code></td>
<td>44 (0x2c)</td>
<td><code>aload_2</code></td>
</tr>
<tr>
<td>21 (0x15)</td>
<td><code>iload</code></td>
<td>45 (0x2d)</td>
<td><code>aload_3</code></td>
</tr>
<tr>
<td>22 (0x16)</td>
<td><code>iload</code></td>
<td>46 (0x2e)</td>
<td><code>iaload</code></td>
</tr>
<tr>
<td>23 (0x17)</td>
<td><code>fload</code></td>
<td>47 (0x2f)</td>
<td><code>iload</code></td>
</tr>
</tbody>
</table>

487
48 (0x30).................................. faload
49 (0x31).................................. daload
50 (0x32).................................. aaload
51 (0x33).................................. baload
52 (0x34).................................. caload
53 (0x35).................................. saload
54 (0x36).................................. istore
55 (0x37).................................. lstore
56 (0x38).................................. fstore
57 (0x39)................................. dstore
58 (0x3a)................................. astore
59 (0x3b)................................. istore_0
60 (0x3c)................................. istore_1
61 (0x3d)................................. istore_2
62 (0x3e)................................. istore_3
63 (0x3f)................................. lstore_0
64 (0x40)................................. lstore_1
65 (0x41)................................. lstore_2
66 (0x42)................................. lstore_3
67 (0x43)................................. fstore_0
68 (0x44)................................. fstore_1
69 (0x45)................................. fstore_2
70 (0x46)................................. fstore_3
71 (0x47)................................. dstore_0
72 (0x48)................................. dstore_1
73 (0x49)................................. dstore_2
74 (0x4a)................................. dstore_3
75 (0x4b)................................. astore_0
76 (0x4c)................................. astore_1
77 (0x4d)................................. astore_2
78 (0x4e)................................. astore_3
79 (0x4f)................................. iastore
80 (0x50)................................. lastore
81 (0x51)................................. fastore
82 (0x52)................................. dastore
83 (0x53)................................. aastore
84 (0x54)................................. bastore
85 (0x55)................................. castore
86 (0x56)................................. sastore
87 (0x57)................................. pop
88 (0x58)................................. pop2
89 (0x59)................................. *dup*
90 (0x5a)............................... *dup_x1*
91 (0x5b)............................... *dup_x2*
92 (0x5c)............................... *dup2*
93 (0x5d)............................... *dup2_x1*
94 (0x5e)............................... *dup2_x2*
95 (0x5f)............................... *swap*
96 (0x60)............................... *iadd*
97 (0x61)............................... *ladd*
98 (0x62)............................... *fadd*
99 (0x63)............................... *dadd*
100 (0x64)............................. *isub*
101 (0x65)............................. *lsub*
102 (0x66)............................. *fsub*
103 (0x67)............................. *dsub*
104 (0x68)............................. *imul*
105 (0x69)............................. *lmul*
106 (0x6a)............................. *fmul*
107 (0x6b)............................. *dmul*
108 (0x6c)............................. *idiv*
109 (0x6d)............................. *ldiv*
110 (0x6e)............................. *fdiv*
111 (0x6f)............................. *ddiv*
112 (0x70)............................. *irem*
113 (0x71)............................. *lrem*
114 (0x72)............................. *frem*
115 (0x73)............................. *drem*
116 (0x74)............................. *inum*
117 (0x75)............................. *lneg*
118 (0x76)............................. *fneg*
119 (0x77)............................. *dneg*
120 (0x78)............................. *ishl*
121 (0x79)............................. *ishl*
122 (0x7a)............................. *ishr*
123 (0x7b)............................. *ishr*
124 (0x7c)............................. *iushr*
125 (0x7d)............................. *lushr*
126 (0x7e)............................. *iand*
127 (0x7f)............................. *land*
128 (0x80)............................. *ior*
129 (0x81)............................. *lor*
130 (0x82)........................... ixor
131 (0x83)........................... lxor
132 (0x84)........................... iinc
133 (0x85)......................... i2l
134 (0x86)........................... i2f
135 (0x87)........................... i2d
136 (0x88)........................... l2i
137 (0x89)........................... l2f
138 (0x8a)........................... l2d
139 (0x8b)........................... f2i
140 (0x8c)........................... f2l
141 (0x8d)........................... f2d
142 (0x8e)........................... d2i
143 (0x8f)........................... d2l
144 (0x90)........................... d2f
145 (0x91)........................... i2b
146 (0x92)........................... i2c
147 (0x93)........................... i2s
148 (0x94)........................... lcmp
149 (0x95)........................... fcmpl
150 (0x96)........................... fcmpg
151 (0x97)........................... dcmpl
152 (0x98)........................... dcmpg
153 (0x99)........................... ifeq
154 (0x9a)........................... ifne
155 (0x9b)........................... iflt
156 (0x9c)........................... ifge
157 (0x9d)........................... ifgt
158 (0x9e)........................... ifle
159 (0x9f)........................... if_icmpeq
160 (0xa0)........................... if_icmpne
161 (0xa1)........................... if_icmplt
162 (0xa2)........................... if_icmpgt
163 (0xa3)........................... if_icmpge
164 (0xa4)........................... if_icmple
165 (0xa5)........................... if_acmpeq
166 (0xa6)........................... if_acmpne
167 (0xa7)........................... goto
168 (0xa8)........................... jsr
169 (0xa9)........................... tables
170 (0xaa)......................... ret
171 (0xab)........................... lookup
172 (0xac)........................... ireturn
173 (0xad)........................... ireturn
174 (0xae)........................... lreturn
175 (0xaf)........................... dreturn
176 (0xb0)........................... areturn
177 (0xb1)........................... return
178 (0xb2)........................... getstatic
179 (0xb3)........................... putstatic
180 (0xb4)........................... getfield
181 (0xb5)........................... putfield
182 (0xb6)........................... invokevirtual
183 (0xb7)........................... invokespecial
184 (0xb8)........................... invokestatic
185 (0xb9)........................... invokevirtual
186 (0xba)........................... xxxunusedxxx
187 (0xbb)........................... new
188 (0xbc)........................... newarray
189 (0xbd)........................... anewarray
190 (0xbe)........................... arraylength
191 (0xbf)........................... athrow
192 (0xc0)........................... checkcast
193 (0xc1)........................... instanceof
194 (0xc2)........................... monitorenter
195 (0xc3)........................... monitorexit
196 (0xc4)........................... wide
197 (0xc5)........................... multianewarray
198 (0xc6)........................... ifnull
199 (0xc7)........................... ifnonnull
200 (0xc8)........................... goto_w
201 (0xc9)........................... jsr_w
254 (0xfe)....................................
255 (0xff)............................

Reserved opcodes:

1 For historical reasons, opcode value 186 is not used.
# character
use in compilation examples, 450
( character
meaning in method descriptor, 51
) character
meaning in method descriptor, 51
/ character
in class and interface names in internal form, 48
; character
meaning in field or method descriptor, 50
< character
meaning in field or method descriptor, 50
< character
in CONSTANT_Methodref_info and CONSTANT_InterfaceMethodref _info names, significance of, 58
in names of <init> and <clinit> methods, 24
> character
in names of <init> and <clinit> methods, 24
[ character
meaning in field or method descriptor, 50

**invokevirtual**, 377
thrown during method resolution, 252

**ACC_ABSTRACT flag**
See also abstract modifier
(access_flags item of ClassFile structure), 44
(access_flags item of method_info structure), 68, 69
(inner_class_access_flags item of InnerClasses_attribute structure), 92

**ACC_BRIDGE flag**
(access_flags item of method_info structure), 69

**ACC_ENUM flag**
(access_flags item of field_info structure), 44, 66, 92

**ACC_FINAL flag**
See also final modifier
(access_flags item of ClassFile structure), 44
(access_flags item of field_info structure), 66
(access_flags item of method_info structure), 69
(inner_class_access_flags item of InnerClasses_attribute structure), 92

**ACC_INTERFACE flag**
See also interfaces
(access_flags item of ClassFile structure), 44
(inner_class_access_flags item of InnerClasses_attribute structure), 92

**ACC_NATIVE flag**
See also native modifier

**aaload instruction**
definition, 262

**aastore instruction**
compilation examples, arrays, 470
constraints, structural, 121
definition, 263

**abrupt completion**
method invocation, 20

**AbstractMethodError**
thrown by
  **invokeinterface**, 366, 367
  **invokespecial**, 370

Index
(access_flags item of method_info structure), 69

**ACC_PRIVATE flag**

*See also* private modifier

(access_flags item of field_info structure), 66

(access_flags item of method_info structure), 69

(inner_class_access_flags item of InnerClasses_attribute structure), 92

**ACC_PROTECTED flag**

*See also* protected modifier

(access_flags item of field_info structure), 66

(access_flags item of method_info structure), 69

(inner_class_access_flags item of InnerClasses_attribute structure), 92

**ACC_PUBLIC flag**

*See also* public modifier

(access_flags item of ClassFile structure), 44

(access_flags item of field_info structure), 66

(access_flags item of method_info structure), 69

(inner_class_access_flags item of InnerClasses_attribute structure), 92

**ACC_STATIC flag**

*See also* static modifier

(access_flags item of field_info structure), 66

(access_flags item of method_info structure), 69

(inner_class_access_flags item of InnerClasses_attribute structure), 92

**ACC_SUPER flag**

*See also* superclasses

(access_flags item of ClassFile structure), 44

**ACC_SYNCHRONIZED flag**

*See also* synchronization

(access_flags item of method_info structure), 69

**ACC_SYNTHETIC flag**

(access_flags item of ClassFile structure), 44, 92

(access_flags item of field_info structure), 66

(access_flags item of method_info structure), 69

**ACC_TRANSIENT flag**

*See also* transient modifier

(access_flags item of field_info structure), 66

**ACC_VARARGS flag**

(access_flags item of method_info structure), 69

**ACC_VOLATILE flag**

*See also* volatile modifier

(access_flags item of field_info structure), 66

**access control**

*See also* access_flags item, IllegalAccessException
during dynamic method lookup

invokeinterface, 366

invokevirtual, 377

enforcement, 252

final fields

putfield, 435

putstatic, 437

instance initialization methods, 24

package private access, 253

**access_flags item**

*See also* access control

(ClassFile structure), 44

(field_info structure), 65

(method_info structure), 68

**aconst_null instruction**

definition, 265

**adding**

double, dadd, 285

float, fadd, 315

int, iadd, 346

long, ladd, 395

**algorithms**

class file verification, 122
conversion of bytes item, CONSTANT_Float_info structure, to float value, 60
conversion of high_bytes and low_bytes items, CONSTANT_Double_info structure, to double value, 62
creation and loading
array classes, 240, 243
classes, 239
interfaces, 240
using a user-defined class loader, 242
using the default class loader, 242
string literals, derivation of, 239
alignment
code array, 77
Java virtual machine instructions, implementation implications, 27
aload instruction
See also astore instruction, wide instruction
constraints, static, 118
definition, 266
aload_<n> instructions
See also astore_<n> instructions
compilation examples
arrays, 469
catching exceptions, 475, 476, 478
compiling finally, 480, 481
invoking methods, 463, 465
operand stack operations, 473
throwing exceptions, 474, 475
working with class instances, 466, 467
constraints, static, 118
definition, 267
ANDing
int, bitwise, iand, 348
long, bitwise, land, 397
anewarray instruction
compilation examples, arrays, 469
constraints, static, 118
definition, 268
AnnotationDefault_attribute structure
(attributes table of method_info structure), 113
areturn instruction
compilation examples
arrays, 470
working with class instances, 466, 467
constraints, structural, 120
definition, 269
arithmetic
adding
double, dadd, 285
float, fadd, 315
int, iadd, 346
long, ladd, 395
ArithmeticException
thrown by idiv, 351
thrown by irem, 381
thrown by ldiv, 405
thrown by lrem, 413
compilation examples, 456
dividing
double, ddiv, 292
float, fdiv, 322
int, idiv, 351
long, ldiv, 405
instruction set, summary, 31
multiplying
double, dmul, 296
float, fmul, 326
int, imul, 362
long, lmul, 408
negating
double, dneg, 298
float, fneg, 328
int, ineg, 363
long, lneg, 409
remainder
double, drem, 299
float, frem, 299
int, irem, 381
long, lrem, 413
subtracting
double, dsub, 304
float, fsub, 334
int, isub, 387
long, lsub, 419
ArithmeticException
thrown by
idiv, 351
irem, 381
ldiv, 405
lrem, 413
array(s)
See also class(es); interfaces; references;
types
ArrayIndexOutOfBoundsException

See also IndexOutOfBoundsException

thrown by

aaload, 262
aastore, 264
baload, 275
bastore, 275
caload, 278
castore, 279
daload, 287
dastore, 288
faload, 317
fastore, 318

attribute_length item

(attribute_info generic structure), 72
(Code_attribute structure), 77
(ConstantValue_attribute structure), 75
(Deprecated_attribute structure), 104
attributes
See also ClassFile structure:
attribute_length item
attribute_name_index item
attributes table
attributes_count item
See also predefined attributes:
AnnotationDefault_attribute
Deprecated_attribute
EnclosingMethod_attribute
Exceptions_attribute
InnerClasses_attribute
LocalVariableTable_attribute
LocalVariableTypeTable_attribute
RuntimeInvisibleAnnotations_attribute
RuntimeInvisibleParameterAnnotations_attribute
RuntimeVisibleAnnotations_attribute
RuntimeVisibleParameterAnnotations_attribute
Signature_attribute
SourceFile_attribute
Synthetic_attribute
defining and naming new, 74
attributes table
(ClassFile structure), 47
(Code_attribute structure), 79
(field_info structure), 67
(method_info structure), 70
attributes_count item
(ClassFile structure), 47
(Code_attribute structure), 79
(field_info structure), 79
(method_info structure), 70

B
B character
meaning in field or method descriptor, 50
backwards branches
structural constraints on instructions, 120
baload instruction
definition, 275
bastore instruction
definition, 276
bibliographic references
 Polling Efficiently on Stock Hardware, 25
big-endian order
bytes item
(CONSTANT_Float_info structure), 60
(CONSTANT_Integer_info structure), 60
class file data storage order, 41
high_bytes and low_bytes items
(CONSTANT_Double_info structure), 62
(CONSTANT_Long_info structure), 61
multibyte characters,
CONSTANT_Utf8_info structure
representation of, 63
binding
See also linking; loading; native modifier
instructions causing
invokeinterface, 366
invokespecial, 370
invokestatic, 374
invokevirtual, 377
of native method implementations, 255
bipush instruction
compilation examples
accessing the runtime constant pool, 457
arrays, 468
constants and local variables in a for loop, 451, 452, 455
invoking methods, 463, 464
while loop, 458
definition, 277
bitwise
ANDing
int, iand, 348
long, land, 395
ORing
int exclusive, ixor, 389
int inclusive, ior, 380
long exclusive, lxor, 421
long inclusive, lor, 412
boolean type
loading from arrays, baload, 275
storing into arrays, bastore, 276
branch
code verification, Pass 3 - bytecode verifier, 226
instruction summary, 36
instructions, constraints, static, 116
int comparison
if_icmp<cond>, 353
with zero, if<cond>, 355
reference comparison
if_acmp<cond>, 352
with null, ifnonnull, 357
with null, ifnull, 358
unconditionally
goto, 338
wide index, goto_w, 339
breakpoint reserved opcode
definition, 258
byte type
boolean array values represented as values of, 12
converting int to, i2b, 340
definition, 8
instruction set handling of, 27
integer arithmetic not directly supported, 31
loading from arrays, baload, 275
pushing, bipush, 277
storing into arrays, bastore, 276
value range, 9
bytes array
(CONSTANT_Utf8_info structure), 65
bytes item
(CONSTANT_Float_info structure), 60
(CONSTANT_Integer_info structure), 60
C
c C character
meaning in field or method descriptor, 50
caload instruction
definition, 278
casting
See also numeric
checkcast, 280
castore instruction
definition, 279
catch clause(s)
ordering of, 478
catch_type item
(Code_attribute structure), 79
char type
arithmetic not directly supported, 31
converting int to, i2c, 341
definition, 8
instruction set handling of, 27
loading from arrays, caload, 278
storing into arrays, castore, 279
value range, 9
checkcast instruction
See also instanceof instruction
constraints, static, 118
definition, 280
checking
types
checkcast, 280
instanceof, 364
**class file format**

See also ClassFile structure
byte storage order, 41
(chapter), 41
data, methods that can read, 41
integrity verification, 122
supported versions, 43

**class file version**
49.0 and above, special rules, 47, 67, 71,
73, 79, 117
50.0 and above, special rules, 73, 123, 482
51.0 and above, special rules, 116

**class loader**
bootstrap, 240
ClassLoader
<clinit> method, as class or interface initialization method, 24, 237
defining, 241
delegating to another, 241
initiating, 241
loading
by a user-defined, 242
by the bootstrap, 242
loading constraints, 244
user-defined, 240

**class(es)**
linking
preparation, 254

**class(es)**
See also array; class file format; class loader; ClassFile structure; interfaces
creation, 240
derivation of symbolic references to at run time, 238
get static fields from, getstatic, 337
initial, specifying to Java virtual machine, 240
instances
uninitialized, structural constraints, 117
libraries, Java virtual machine support for, 38
names, name_index item
(CONSTANT_Class_info structure) as reference to, 56
put into static fields, putstatic, 437
static methods
invocation instruction summary, 37

invoking, invokestatic, 374
types
as reference type, 12

**class_index item**
(CONSTANT_Fieldref_info structure), 58
(CONSTANT_InterfaceMethodref_info structure), 58
(CONSTANT_Methodref_info structure), 58

ClassCastException
thrown by checkcast, 281

ClassCircularityError
thrown during
class or interface loading, 246
class or interface resolution, 246

**classes array**
(InnerClasses_attribute_info structure), 90

**ClassFile structure**
See also ClassFile substructures:
access_flags item
attributes table
attributes_count item
classloader table
classfile format, 497
constant_pool table
constant_pool_count item
classfile offset structure
classloader table
constant_pool_count item
field_info structure
fields table
fields_count item
interfaces array
interfaces_count item
magic item
major_version item
method_info structure
methods table
methods_count item
minor_version item
super_class item
this_class item
constant_pool_table, Java virtual machine representation, 15

format
ability to read as Java virtual machine implementation requirement, 7
as overview, 7
integrity verification, 122
syntax and item descriptions, 41
<clinit> method
as class or interface initialization method
   name, 24
constant_pool table, reference to, 58
invocation of, static constraints, 117
name_index item (method_info structure)
   reference, 70

code
   See also code array; Code_attribute
   structure
blocks, synchronization, instruction
   summary, 37
code array
   (Code_attribute structure)
   constraints, static, 115
   constraints, structural, 119
   size and location, 76
data-flow analysis, 123
code generation
   See also binary, compatibility; compile-time
   errors; exceptions; optimization
Code_attribute structure
   (attributes table of method_info
   structure), 76
code_length item
   (Code_attribute structure), 77
comparisons
double, dcmp<op>, 289
float, fcmp<op>, 319
int
   if_icmp<cond>, 353
   with zero, if<cond>, 355
long, lcmp, 399
numerical
   floating-point positive and negative
   zero, 11
   implications of unordered NaN
   values, 11
reference
   if_acmp<cond>, 352
   with null, ifnull, 357, 358
compilation
for the Java virtual machine, (chapter), 449
Java virtual machine assembly language,
   format, 450
completion
method invocation
   abrupt, 20
   normal, 20
computational type
   definition, 30
conditional
   See also control flow
   branch, instruction summary, 36
CONSTANT_Class_info structure
class names referenced from, 48
   (constant_pool table), items and
   meaning, 56
derivation of symbolic reference from at run
   time, 238
super_class item, as ClassFile structure
   reference to a, 45
this_class item, as ClassFile structure
   reference to a, 45
CONSTANT_Class tag
   (CONSTANT_class_info structure), 56
CONSTANT_Double_info structure
   (constant_pool table), items and
   meaning, 61
derivation of constant value from at run
   time, 239
CONSTANT_Double tag
   (CONSTANT_Double_info structure), 61
CONSTANT_Fieldref_info structure
   (constant_pool table), items and
   meaning, 58
derivation of symbolic reference from at run
   time, 238
CONSTANT_Fieldref tag
   (CONSTANT_Fieldref_info structure), 57
CONSTANT_Float_info structure
   (constant_pool table), items and
   meaning, 60
derivation of constant value at run
   time, 239
CONSTANT_Float tag
   (CONSTANT_Float_info structure), 59
CONSTANT_Integer_info structure
   (constant_pool table), items and
   meaning, 60
derivation of constant values at run
   time, 239
CONSTANT_Integer tag
   (CONSTANT_Integer_info structure), 59
CONSTANT_InterfaceMethodref_info
   structure
   (constant_pool table), items and
   meaning, 58
derivation of symbolic reference from at run time, 238

CONSTANT_InterfaceMethodref tag
(CONSTANT_InterfaceMethodref_info structure), 57

CONSTANT_Long_info structure
(constant_pool table), items and meaning, 61
derivation of constant value at run time, 239

CONSTANT_Long tag
(CONSTANT_Long_info structure), 61

CONSTANT_Methodref_info structure
(constant_pool table), items and meaning, 58
derivation of symbolic reference from at run time, 238

CONSTANT_Methodref tag
(CONSTANT_Methodref_info structure), 57

CONSTANT_NameAndType_info structure
class names referenced from, 48
(constant_pool table), items and meaning, 63
derivation of symbolic reference from at run time, 239

CONSTANT_NameAndType_tag
(CONSTANT_NameAndType_info structure), 62

constant_pool_count item
(ClassFile structure), 43

constant_pool table
(ClassFile structure), 43, 55
constantvalue_index item values (table), 75
derivation of symbolic references from at run time, 237
tag values (table), 56

CONSTANT_String_info structure
(constant_pool table), items and meaning, 59
derivation of symbolic reference from at run time, 239

CONSTANT_String tag
(CONSTANT_String_info structure), 59

CONSTANT_Utf8_info structure
attribute_name_index item
run time, 238
items and meaning, 58
CONSTANT_Float_info structure
derivation of constant value from at run
time, 239
items and meaning, 60
CONSTANT_Integer_info structure
derivation of constant value from at run
time, 239
items and meaning, 60
CONSTANT_InterfaceMethodref_info
structure
derivation of symbolic reference from at
run time, 238
items and meaning, 58
CONSTANT_Long_info structure
derivation of constant value from at run
time, 239
items and meaning, 61
CONSTANT_Methodref_info structure
derivation of symbolic reference from at
run time, 238
items and meaning, 58
CONSTANT_NameAndType_info structure
indirect use of at run time, 239
items and meaning, 63
CONSTANT_String_info structure
derivation of symbolic reference from at
run time, 239
items and meaning, 59
CONSTANT_Utf8_info structure
descriptor_index item,
CONSTANT_NameAndType_info
reference, 63
indirect use of at run time, 240
items and meaning, 63, 93
ConstantValue_attribute structure
field_info structure value, 67
support required for, 74
floating-point
double, CONSTANT_Double_info structure
representation, 61
load and store instructions, summary, 30
pushing
double, dconst_<d>, 291
float, fconst_<f>, 321
int, icont_<i>, 350
ldc, 401
long, lconst_<l>, 400
wide index, ldc_w, 402
runtime constant pool, 15
derivation of, 237
frame reference, dynamic linking supported
by, 17, 19
ConstantValue_attribute structure
(attributes table of field_info
structure), 74
(field_info structure), 75
constantvalue_index structure
(ConstantValue_attribute
structure), 75
constraints
class loading, 244
enforcement of, by class file verifier, 257
Java virtual machine code
static, specification of, 115
structural, specification of, 119
operand stack manipulation, 19
constructors
as instance initialization method, 24
control flow
See also threads
branch on
int comparison with zero, if<cond>, 355
int comparison, if_icmp<cond>, 353
reference comparison with null,
ifnonnull, 357
reference comparison with null,
ifnull, 358
reference comparison,
if_acmp<cond>, 352
compilation examples, for keyword, 452
compilation of, while keyword, 458
instruction summary, 36
unconditional goto
goto, 338
wide index, goto_w, 339
conversions
bytes item, CONSTANT_Float_info structure, algorithm, 60
narrowing primitive
  double to float, d2f, 282
  double to int, d2i, 283
  double to long, d2l, 284
  float to int, f2i, 313
  float to long, f2l, 314
  impact on precision, 34
  int to byte, i2b, 340
  int to char, i2c, 341
  int to short, i2s, 345
  long to int, l2i, 394
  support for, 34

widening primitive
  float to double, f2d, 312
  impact on numeric precision, 33
  int to double, i2d, 342
  int to float, i2f, 343
  int to long, i2l, 344
  long to double, l2d, 392
  long to float, l2f, 393
  support for, 33

cp_info structure
  (generic form of items in the
   constant_pool table), 55
  tag values (table), 56

creating
  array classes, 240, 243
  arrays
    multidimensional, multianewarray, 426
    primitive type, newarray, 430
    reference type, anewarray, 268
  class instances
    instruction summary, 35
    new, 428
  classes and interfaces, 240

current
  class, 17
  frame, 17
  method, 17

D

D character
  meaning in field or method descriptor, 50

d2f instruction
  definition, 282

d2i instruction
  definition, 283

d2l instruction
  definition, 284

dadd instruction
  compilation examples
    constants and local variables in a for
      loop, 453, 455
    while loop, 459
  definition, 285

daload instruction
  definition, 287

dastore instruction
  definition, 288

data
  areas, runtime
    constant pool, 15
    heap, 14
    Java virtual machine stack, 13
    method area, 15
    native method stacks, 16
    pc register, 13
  types, Java virtual machine, 7

data types
  arguments, structural constraints on
    instructions, 119
  checking
    checkcast, 280
    instanceof, 364
  conversion
    instructions, 33
  Java virtual machine
    instruction set encoding of, 27
    mapping between storage types and computa-
      tional types (table), 30
    support for (table), 29

data-flow analysis
  code array, 123
  running, Pass 3 - bytecode verifier, 228

dcmp<op> instructions
  compilation examples
    constants and local variables in a for
      loop, 453
    while loop, 459, 461
  compilation examples, while loop, 461
  definition, 289
**dconst_<d> instructions**

compilation examples
- constants and local variables in a for loop, 453
- while loop, 459
- definition, 291

**ddiv instruction**
- definition, 292

**debugging**
- breakpoint reserved opcode, 258
- Java virtual machine implementation
- issues, 39

**defineClass method**
- ClassLoader class, creation of classes and interfaces by, 243

**delegation**
- to another class loader, 241

**Deprecated_attribute structure**
- (attributes table of Classfile, field_info, method_info structures), 103

**descriptor_index item**
- (CONSTANT_NameAndType_info structure), 63
- (field_info structure), 67
- (LocalVariableTable_attribute structure), 99, 101
- (method_info structure), 68

**descriptors**
- characteristics and use, 48
- field
  - structural constraints on instructions, 119, 120
  - syntax and item descriptions, 49
  - as value of CONSTANT_Utf8_info structure referenced by descriptor_index item, CONSTANT_NameAndType_info structure, 63
  - as value of CONSTANT_Utf8_info structure referenced by descriptor_index item, field_info structure, 67
- grammar for specification of, 49
- method
  - argument number limitation, 235
  - syntax and item descriptions, 51
  - as value of CONSTANT_Utf8_info structure referenced by descriptor_index item, CONSTANT_NameAndType_info structure, 63

**dividing**
- double, ddiv, 292
- float, fdiv, 322
- int, idiv, 351
- long, ldiv, 405

**dload instruction**
- constraints, static, 118
- definition, 294

**dload_<n> instructions**
- compilation examples
  - constants and local variables in a for loop, 453, 455
  - constraints, static, 118
  - definition, 295

**dmul instruction**
- definition, 296

**dneg instruction**
- definition, 298

**double type**
- See also floating-point
- adding, dadd, 285
- comparing, dcmp<op>, 289
- compilation examples, 453
- converting
  - float to, f2d, 312
  - int to, i2d, 342
  - long to, l2d, 392
  - to float, d2f, 282
  - to long, d2l, 284
- definition, 8
- dividing, ddiv, 292
- double value set, 9
- double-extended-exponent value set, 10
- field descriptor specification, 49
- loading from
  - arrays, daload, 287
  - local variables, dload, 294
  - local variables, dload_<n>, 295
- multiplying, dmul, 296
- negating, dneg, 298
- pushing constants, dconst_<d>, 291
- pushing, wide index, ldc2_w, 404
- remainder, drem, 299
- representation in constant pool, 62
double value set • exception_index_table array

returning from method invocation, \textit{dreturn}, 301

storing into
arrays, \textit{dastore}, 288
local variables, \textit{dstore}, 302
local variables, \textit{dstore_{<n>}}, 303

\textbf{double value set}

definition, 9
parameters (table), 10

\textbf{Double_variable_info structure}, 87

\textbf{double-extended-exponent value set}

definition, 10
parameters (table), 10

\textit{drem} instruction

definition, 299

\textit{dreturn} instruction

compilation examples, constants and local variables in a \texttt{for} loop, 455
constraints, structural, 119
definition, 301

\textit{dstore} instruction

compilation examples, accessing the runtime constant pool, 458
constraints, static, 118
definition, 302

\textit{dstore_{<n>}} instructions

compilation examples
constants and local variables in a \texttt{for} loop, 453
\texttt{while} loop, 459
constraints, static, 118
definition, 303

\textit{dsub} instruction

definition, 304

\textbf{dup} instruction

compilation examples
arrays, 470
operand stack operations, 473
throwing exceptions, 474
working with class instances, 466
definition, 305

\textbf{dup instructions}

operand stack manipulation constraints, 19

\textbf{dup_{x1}} instruction

definition, 306

\textbf{dup_{x2}} instruction

definition, 307

dup2 instruction
definition, 308

dup2_{x1} instruction

compilation examples
operand stack operations, 473
definition, 309

dup2_{x2} instruction

definition, 310

duplicating

See also \textit{dup} instructions
operand stack value(s)
dup, 305
dup_{x1}, 306
dup_{x2}, 307
dup2, 308
dup2_{x1}, 309
dup2_{x2}, 310

\textbf{E}

\textbf{EnclosingMethod_attribute structure}

(attributes table of ClassFile structure), 92

\textbf{end_pc item}

(Code_attribute structure), 76

\textbf{entering}

See also locks; monitor
monitor for object, \textit{monitorenter}, 422

\textbf{Error}

thrown by
getstatic, 337
invokestatic, 376
new, 429
putstatic, 438

\textbf{errors}

heap-related, OutOfMemoryError, 15
Java virtual machine stack-related
OutOfMemoryError, 14
StackOverflowError, 14
method area-related,

OutOfMemoryError, 15
native method stack-related
OutOfMemoryError, 17
StackOverflowError, 17
throwing, \textit{athrow}, 273

\textbf{exception_index_table array}

(Exceptions_attribute structure), 88

\textbf{E}

\textbf{EnclosingMethod_attribute structure}

(attributes table of ClassFile structure), 92

\textbf{end_pc item}

(Code_attribute structure), 76

\textbf{entering}

See also locks; monitor
monitor for object, \textit{monitorenter}, 422

\textbf{Error}

thrown by
getstatic, 337
invokestatic, 376
new, 429
putstatic, 438

\textbf{errors}

heap-related, OutOfMemoryError, 15
Java virtual machine stack-related
OutOfMemoryError, 14
StackOverflowError, 14
method area-related,

OutOfMemoryError, 15
native method stack-related
OutOfMemoryError, 17
StackOverflowError, 17
throwing, \textit{athrow}, 273

\textbf{exception_index_table array}

(Exceptions_attribute structure), 88

\textbf{E}

\textbf{EnclosingMethod_attribute structure}

(attributes table of ClassFile structure), 92

\textbf{end_pc item}

(Code_attribute structure), 76

\textbf{entering}

See also locks; monitor
monitor for object, \textit{monitorenter}, 422

\textbf{Error}

thrown by
getstatic, 337
invokestatic, 376
new, 429
putstatic, 438

\textbf{errors}

heap-related, OutOfMemoryError, 15
Java virtual machine stack-related
OutOfMemoryError, 14
StackOverflowError, 14
method area-related,

OutOfMemoryError, 15
native method stack-related
OutOfMemoryError, 17
StackOverflowError, 17
throwing, \textit{athrow}, 273

\textbf{exception_index_table array}

(Exceptions_attribute structure), 88
exception_table array
(Code_attribute structure), 76

exception_table_length item
(Code_attribute structure), 76

exceptions
asynchronous, causes and handling of, 25
errors
ExceptionInInitializerError, 255
NoClassDefFoundError, 254
OutOfMemoryError, 255

exceptions
abrupt completion, 20
dispatching, frame use for, 17
handling
by Java virtual machine, 25
 instruction summary, 37
structural constraints on instructions, 226
normal completion, characterized by lack of, 20
requirements for throwing, 79
throwing, athrow, 273

Exceptions_attribute structure
(attributes table of method_info structure), 88

execution
paths, structural constraints on
instructions, 119

exiting
See also Java virtual machine; locks; moni-
tor
monitor for object, monitorexit, 424

extend
local variable index by additional bytes,
wide, 447

f

F character
meaning in field or method descriptor, 50

f2d instruction
definition, 312

f2l instruction
definition, 314

fold instruction
definition, 315

fields
class, field_info structure access
flags, 65
constant pool references, verification
process, 226
creation and manipulation, instruction
summary, 35
derivation of symbolic references to at run
time, 238
descriptor
syntax and meaning, 49
as value of CONSTANT_Utf8_info struc-
ture referred by descriptor_index item,
CONSTANT_NameAndType_info
structure, 63
get from class instances, getfield, 335
interface, field_info structure access
flags, 65
lookup, 250
number limitation, 235
protected structural constraints, 120
put into class instances, putfield, 435
references, resolution, 250
resolution, 250
static
get from classes, getstatic, 337
put into classes, putstatic, 437
types, 50

fields table
(ClassFile structure), 46

fields_count item
(ClassFile structure), 46

final modifier
class
enforcement, 123, 377
finally clause

data-flow analysis during class file verification, 234
implementation of
  in catch_type item (Code_attribute structure), 79
try-finally clause, Sun’s Java compiler output characteristics, 232
uninitialized object restrictions, Pass 3 - bytecode verifier, 232

findSystemClass method
ClassLoader class, loading of classes and interfaces by, 243

float instruction
See also wide instruction
constraints, static, 118
definition, 324

float_<n> instructions
constraints, static, 118
definition, 325

float type
See also floating-point
adding, fadd, 315
comparing, fcmp<op>, 320
converting
double to, d2f, 282
int to, i2f, 343
long to, l2f, 393
to double, f2d, 312
to int, f2i, 313
to long, f2l, 314
dividing, fdiv, 322
float value set, 9
float-extended-exponent value set, 10
loading from
  arrays, faload, 317
  local variables, float, 324
  local variables, float_<n>, 325
multiplying, fmul, 326
negating, fneg, 328
pushing constants, fconst_<f>, 321
remainder, frem, 329
representation in constant pool, 60
returning from method invocation,
  freturn, 331
storing into
  arrays, fastore, 318
  local variables, fstore, 332
  local variables, fstore_<n>, 333
subtracting, fsub, 334

float value set
definition, 9
parameters (table), 10

float_variable_info structure, 86

float-extended-exponent value set
definition, 10
parameters (table), 10

floating-point
comparison, IEEE 754 conformance, 33, 36
types
  components, and values, 9
underflow and overflow, Java virtual machine handling, 32

fmul instruction
definition, 326

fneg instruction
definition, 328

for keyword
compilation examples, 451

Format checking, 115

forward slashes (/)
in class and interface names in internal form, 48

FP-strict floating point mode
definition, 22

frames
See also stacks
definition, 17
exception handling impact on, 26
local variables, 18

frem instruction
definition, 329

freturn instruction
constraints, structural, 120
definition, 331

fstore instruction
constraints, static, 118
definition, 332

fstore_<n> instructions
constraints, static, 118
G

garbage collection
algorithm, not specified by Java virtual machine specification, 7
as implementation of automatic storage management system, 14

Generic types, 52

getfield instruction
compilation examples
operand stack operations, 473
working with class instances, 467
constraints
static, 117
structural, 120
definition, 335

getstatic instruction
constraints, static, 117
definition, 337

goto instruction
compilation examples
compiling finally, 480
constants and local variables in a for loop, 451, 453, 455
while loop, 458, 459
constraints, static, 116
definition, 338, 339

goto_w instruction
constraints, static, 116
definition, 339

gradual underflow
conformance
add double, dadd, 285
add float, fadd, 315
dividing
double conformance, ddiv, 292
float conformance, fddiv, 322
multiplying
double conformance, dmul, 296
float conformance, fmul, 326
subtracting
double conformance, dsub, 304
float conformance, fsub, 334

grammar
descriptor specification, 49

H

handler_pc item
(exception_table array of Code_attribute structure), 78

handling an exception
by Java virtual machine, 25

hash sign (#)
use in compilation examples, 450

heap
definition, 14
definition, OutOfMemoryError, 15

high_bytes item
(CONSTANT_Double_info structure), 62

high_bytes item
(CONSTANT_Long_info structure), 61

I

I character
meaning in field or method descriptor, 50

i2b instruction
definition, 340

i2c instruction
definition, 341

i2d instruction
definition, 342

i2f instruction
definition, 343

i2l instruction
definition, 344

i2s instruction
compilation examples, constants and local variables in a for loop, 455
definition, 345

iadd instruction
compilation examples
arithmetic, 456
constants and local variables in a for loop, 455
receiving arguments, 462
definition, 346

iaload instruction
compilation examples, arrays, 469
**iand instruction**
- compilation examples, arithmetic, 456
- definition, 347

**istore instruction**
- compilation examples, arrays, 469
- definition, 349

**iconst_<i> instructions**
- compilation examples
  - arithmetic, 456
  - arrays, 470
  - compiling switches, 471
  - constants and local variables in a for loop, 451
  - operand stack operations, 473
  - while loop, 458, 461
- definition, 350

**idiv instruction**
- definition, 351

**IEEE 754 standard**
- bibliographic reference, 21
- comparing
  - double conformance, *dcmp<op>* , 289
  - float conformance, *fcmp<op>* , 319
- conformance
  - add double *dadd*, 285
  - add float *fadd*, 315
- dividing
  - double conformance, *ddiv*, 292
  - float conformance, *fdiv*, 322
- floating-point
  - double bit layout, high_bytes and low_bytes items,
    CONSTANT_Double_info structure, 61
- operation conformance to, 32
- key differences between Java virtual machine and, 21
- multiplying
  - double conformance, *dmul*, 296
  - float conformance, *fmul*, 326
- remainder
  - *drem* not the same as, *drem*, 299
  - *frem* not the same as, *frem*, 329
- subtracting
  - double conformance, *dsub*, 304
  - float conformance, *fsub*, 334

**if<cond> instructions**
- compilation examples
  - constants and local variables in a for loop, 453
  - throwing exceptions, 474
  - while loop, 460, 461
- constraints, static, 116
- definition, 355

**if_acmp<cond> instructions**
- constraints, static, 116
- definition, 352

**if_icmp<cond> instructions**
- compilation examples
  - constants and local variables in a for loop, 451, 453, 455
  - while loop, 458
- constraints, static, 116
- definition, 353

**ifnonnull instruction**
- constraints, static, 116
- definition, 357

**ifnull instruction**
- compilation examples, working with class instances, 466
- constraints, static, 116
- definition, 358

**iinc instruction**
- compilation examples
  - constants and local variables in a for loop, 451, 452
  - while loop, 458
- constraints, static, 118
- definition, 359

**IllegalAccessError**
- thrown by
  - invokeinterface, 368
  - multianewarray, 427
  - putfield, 436
  - putstatic, 438
- thrown during
  - class or interface resolution, 250
  - field resolution, 251
  - method resolution, 252

**IllegalMonitorStateException**
- thrown by
  - areturn, 269
  - athrow, 274
  - dreturn, 301
iload instruction
See also istore instruction; wide instruction
constraints, static, 118
definition, 360
iload_<n> instructions
See also istore_<n> instructions
compilation examples
arithmetic, 456
arrays, 469
compiling switches, 471, 472
constants and local variables in a for
loop, 451, 452, 455
receiving arguments, 462
throwing exceptions, 474
while loop, 459
working with class instances, 467
consstraints, static, 118
definition, 361
impdep1 reserved opcode
definition, 258
impdep2 reserved opcode
definition, 258
implementation
attributes
optional, handling, 70
predefined, support requirements, 70
considerations
exception handling, 89
frames, extensions permitted, 20
heap, 14
Java virtual machine stack, 13
method area, 15
native method stacks, 16
operand stacks, 18
runtime constant pool, 15
constraint enforcement strategies, 257
constraints
Java virtual machine code, static, 115
Java virtual machine code, structural, 119
implications, opcode design and
alignment, 26
Java virtual machine, strategies and
requirements, 39
object representation, 20
requirements and non-requirements, 7
Sun's JVM implementation
boolean arrays as byte arrays, 12, 275
class file format, versions supported, 43
classloading behavior, 243
imul instruction
definition, 362
IncompatibleClassChangeError
thrown by
getfield, 335
getstatic, 337
invokeinterface, 368
invokespecial, 372
invokestatic, 375
invokevirtual, 379
putfield, 436
putstatic, 438
thrown during
class or interface loading, 245
class or interface resolution, 246
interface method resolution, 252
method resolution, 251
increment
local variable by constant, iinc, 359
index item
(LocalVariableTable_attribute
structure), 99, 101
info array
(attribute_info generic structure), 72
<init> method
constant_pool reference to, 58
as instance initialization method name, 24
invocation of
static constraints, 117
structural constraints, 119
method_info structure access flags, 68
name_index item (method_info)
reference, 70
initial class
definition, 240
initialization
See also <clinit> method; <init> method
(chapter), 237
class or interface, reasons for
getstatic, 253
initial class, 253
initialization of a subclass, 253
**invokestatic**, 253
**new**, 253
**putstatic**, 253
**reflection**, 253
instance, data-flow analysis during **class file** verification, 230
method
class or interface (**<clinit>**), 24
instance (**<init>**), 24
when initiated, 253
**inner_class_access_flags item**
(classes array of
**InnerClasses_attribute** structure), 91
**inner_class_info_index item**
(classes array of
**InnerClasses_attribute** structure), 91
**inner_name_index item**
(classes array of
**InnerClasses_attribute** structure), 91
**InnerClasses_attribute structure**
(attributes table of **ClassFile** structure), 89
**instanceof** instruction
definition, 364
**instances**
*See also* array
creating
**new**, 428
creation
instruction summary, 35
determining if an object is a particular type,
**instanceof**, 364
enter monitor for, **monitorenter**, 422
exiting monitor for, **monitorexit**, 424
field descriptor specifications, 51
getting values of fields from, **getfield**, 335
initialization
data-flow analysis during **class file** verification, 231
**field_info** structure access flags, 65
structural constraints on instructions, 119
**instanceof** instruction, constraints,
static, 118
Java virtual machine support for, 8
manipulation, instruction summary, 35
methods
accessing, structural constraints on instructions, 120
data-flow analysis during **class file** verification, 231
invoking, instruction summary, 36
invoking, **invokespecial**, 370
invoking, **invokevirtual**, 377
**method_info** structure access flags, 68
putting values of fields into, **putfield**, 435
reference type relationship to, 8
uninitialized, restrictions, Pass 3 - bytecode verifier, 231
variables
accessing, structural constraints on instruction, 120
**getfield**, 335
putting fields into **putfield**, 435
**InstantiationError**
thrown by **new**, 428
**instructions**
constraints, static, 115
Java virtual machine instruction set execution loop, 26
format, 7
load summary, 30
operands, verification process, 227
**set**
arithmetic, summary, 31
notation for families of, 31
summary, 26
type encoding limitations of, 27
**int type**
adding, **iadd**, 346
ANDing, bitwise, **iand**, 348
branch int comparison
**if_icmp<cond>**, 353
with zero, **if<cond>**, 355
converting
double to, **d2i**, 283
**f2i**, 313
to byte, **i2b**, 340
to char, **i2c**, 341
to double, **i2d**, 342
to float, **i2f**, 342, 343
to long, **i2l**, 344
to short, *i2f*, 345

definition, 8
dividing, *idiv*, 351
instruction set handling of, 27
loading from
arrays, *iaload*, 347
local variables, *iload*, 360
local variables, *iload_n*, 361
multiplying, *imul*, 362
negating, *ineg*, 363
ORing
  bitwise, exclusive, *ixor*, 389
  bitwise, inclusive, *ior*, 380
pushing constants, *iconst_n*, 350
remainder, *irem*, 381
returning from method invocation,
  *ireturn*, 382
shift left, arithmetic, *ishl*, 383
shift right
  arithmetic, *ishr*, 384
  logical, *iushr*, 388
storing into
arrays, *iastore*, 349
local variables, *istore*, 385
local variables, *istore_n*, 386
subtracting, *isub*, 387
value range, 9

**Integer_variable_info structure**, 86

**integral**
types
  definition, 8
  values, 9

**interfaces**
creation, 240
derivation of symbolic references to at run time, 238
methods
derivation of symbolic references to at run time, 238
invocation instruction summary, 36
invoking, *invokeinterface*, 366
method_info structure access flags, 68
resolution, 252
types
  as reference type, 12

**interfaces array**
(ClassFile structure), 46

**interfaces_count item**
(ClassFile structure), 46

**intern method**
String class, 239

**InternalError**
as Java Virtual machine error, 258

**invokeinterface instruction**
constraints, static, 117
definition, 366

**invokespecial instruction**
See also ACC_SUPER modifier
access flag use to select alternative semantics, 45
compilation examples
  arrays, 470
  invoking methods, 465
  throwing exceptions, 474
  working with class instances, 466
constraints
  static, 117
  structural, 119
definition, 370
instance initialization by, 24

**invokestatic instruction**
compilation examples, invoking methods, 464
constraints, static, 117
definition, 374

**invokevirtual instruction**
compilation examples
  catching exceptions, 475, 476, 477
  compiling finally, 480, 481
  invoking methods, 463
  throwing exceptions, 474, 475
  working with class instances, 467
constraints, static, 117
definition, 377

**invoking**
methods
  class, *invokestatic*, 374
  instance, *invokespecial*, 370
  instance, *invokevirtual*, 377
  interface, *invokeinterface*, 366

**ior instruction**
definition, 380

**irem instruction**
declaration, 381

**ireturn instruction**
compilation examples
  arithmetic, 457
  compiling switches, 471, 472
invoking methods, 463, 464, 465
receiving arguments, 462
while loop, 460
constraints, structural, 120
definition, 382

ishl instruction
definition, 383

ishr instruction
definition, 384

istore instruction
See also iload instruction
constraints, static, 118
definition, 385

istore_<n> instructions
See also iload_<n> instructions
compilation examples
accessing the runtime constant pool, 458
arrays, 469
constants and local variables in a for
loop, 451, 455
while loop, 459
constraints, static, 118
definition, 386

isub instruction
compilation examples, arithmetic, 456
definition, 387

items
class file items, 41

jsxh instruction
definition, 388

ixor instruction
compilation examples, arithmetic, 457
definition, 389

Java virtual machine stack
definition, 13
Java, compiler, 52
JIT (just-in-time) code generation
Java virtual machine implementation
issues, 40, 449

jsr instruction
compilation examples, compiling
finally, 479, 481
constraints
static, 116
structural, 120, 121
definition, 390
returnAddress type used by, 12
try–finally clause implementation use,
Sun’s Java compiler output
characteristics, 232

jsxw instruction
constraints
static, 116
structural, 116
definition, 391
returnAddress type used by, 12

jump table
access
by index and jump, tableswitch, 445
by key match and jump,
lookupswitch, 410
alignment concerns, 77

jump to subroutine instructions
constraints, static, 116
jsx, 390
wide index, jsxw, 391

JVM
See Java virtual machine

J

J character
meaning in field or method descriptor, 50

Java programming language
concepts, (chapter), 5

Java virtual machine
assembly language, format, 450
compiling for, (chapter), 449
life cycle, 240
startup, 240
structure of (chapter), 7

L

L character
meaning in field or method descriptor, 50

L<classname>;
meaning in field or method descriptor, 50

l2d instruction
definition, 392

l2f instruction
definition, 393
**ladd instruction**
compilation examples, operand stack operations, 473
definition, 395

**laload instruction**
definition, 396

**land instruction**
definition, 397

**lastore instruction**
definition, 398

**lcmp instruction**
definition, 399

**lconst_<t> instructions**
compilation examples
accessing the runtime constant pool, 458
operand stack operations, 473
definition, 400

**ldc instruction**
compilation examples, accessing the runtime constant pool, 457
constraints, static, 117
definition, 401

**ldc_w instruction**
constraints, static, 117
definition, 402

**ldc2_w instruction**
compilation examples
accessing the runtime constant pool, 458
constants and local variables in a for loop, 453
while loop, 460
constraints, static, 117
definition, 404

**ldiv instruction**
definition, 405

**left angle bracket <**
in CONSTANT_Methodref_info and
CONSTANT_InterfaceMethodref_info names, significance of, 58

**left parentheses (**
meaning in method descriptor, 51

**left square bracket [**
meaning in field or method descriptor, 50

**length item**
(CONSTANT_Utf8_info structure), 63
(LocalVariableTable_attribute structure), 99, 101

**limitations**
Java virtual machine, 234

**line_number item**
(line_number_table array of LineNumberTable_attribute structure), 98

**line_number_table**
(LineNumberTable_attribute structure), 98

**line_number_table_length item**
(LineNumberTable_attribute structure), 98

**LineNumberTable_attribute structure**
(attributes table of Code attribute), 97

**linking**
See also binding; preparation; resolution; verification
(chapter), 237
class files verification issues, 122
definition, 247
dynamic, frame use for, 17

**literals**
strings, resolution of, 239

**lload instruction**
definition, 406

**lload_<n> instructions**
definition, 407

**lmul instruction**
definition, 408

**lneg instruction**
definition, 409

**loadClass method**
ClassLoader class, loading of classes and interfaces by, 243

**loading**
See also class loader; linking; verification
(chapter), 237
class or interface, 240
class or interface, errors ClassCircularityError, 246
IncompatibleClassChangeError, 246
NoClassDefFoundError, 242, 245
constraints, 244
deblegation, 241
from arrays of type byte or boolean, baload, 275
char, caload, 278
double, daload, 287
float,Faload, 317
int, iload, 347
long, laload, 394
local_variable_table array • long type

double, dstore, 302
double, dstore_<n>, 303
float, fstore, 332
float, fstore_<n>, 333
int, istore, 385
int, istore_<n>, 386
long, lstore, 417
long, lstore_<n>, 418
reference, astore, 271
reference, astore_<n>, 272

LocalVariableTable_attribute structure (attributes table of Code attribute), 99
LocalVariableTypeTable_attribute structure (attributes table of Code attribute), 101
locks
See also IllegalMonitorStateException; monitors; threads (chapter), 485
ACC_SYNCHRONIZED flag, method_info structure, 69
structured use of, 37

long type
adding, ladd, 395
ANDing, bitwise, land, 397
comparing, lcmp, 399
constant, CONSTANT_Long_info structure representation, syntax and item descriptions, 61
converting
double to, d2l, 284
float to, f2l, 314
int to, i2l, 344
to double, l2d, 392
to float, l2f, 393
to int, l2i, 394
definition, 8
dividing, ldiv, 405
loading
double, astore, 302
from arrays, laload, 396
from local variables, lload, 406
from local variables, lload_<n>, 407
multiplying, lmul, 408
negating, lneg, 409
ORing
      bitwise, exclusive, lxor, 421
      bitwise, inclusive, lor, 412
pushing

local_variable_table array
(LocalVariableTable_attribute structure), 99, 101
local_variable_table_length item
(LocalVariableTable_attribute structure), 99, 101

local variables
See also parameters; variables
accessing, structural constraints on instructions, 120
compilation examples, 451
data-flow analysis, 226
definition, 18
exception handling impact on, 26
instructions for accessing more, summary, 30
load and store, summary, 30
specialized to handle, advantages of, 452
loading from
double, dload, 294
double, dload_<n>, 295
float, fload, 324
float, fload_<n>, 325
int, iload, 360
int, iload_<n>, 361
long, lload, 406
long, lload_<n>, 407
reference, aload, 266
reference, aload_<n>, 267
location of, 100, 102
maximum number, 77
reuse, advantages of, 452
states, merging, during data-flow analysis, 229
storing into
Long_variable_info structure, 86 • methods

const, lconst_<l>, 400
wide index, ldc2_w, 404
remainder, lrem, 413
returning from method invocation,
  lreturn, 414
shift left, lshl, 415
shift right
  arithmetic, lshr, 416
  logical, lushr, 417
storing into
  arrays, lastore, 398
  local variables, lstore, 417
  local variables, lstore_<n>, 418
subtracting, lsub, 419
value range, 9
Long_variable_info structure, 86
lookupswitch instruction
  See also tableswitch instruction
code array alignment effect, 77
compilation examples, compiling
  switches, 472
cons, static, 116
definition, 410
lor instruction
definition, 412
low_bytes item
  (CONSTANT_Double_info structure), 61
  (CONSTANT_Long_info structure), 61
lrem instruction
definition, 413
lreturn instruction
  compilation examples, operand stack
    operations, 473
cons, structural, 120
definition, 414
lshl instruction
definition, 415
lshr instruction
definition, 416
lstore instruction
cons, static, 118
definition, 417
lstore_<n> instructions
  compilation examples, accessing the run-
    time constant pool, 458
cons, static, 118
definition, 418
lsub instruction
definition, 419
lushr instruction
definition, 420
lushr instruction
definition, 421

M
magic item
  (ClassFile structure), 43
magic number
  See also magic item
main method
  invocation of on startup, 240
major_version item
  (ClassFile structure), 43
mapping
  symbolic references to concrete values, as
    part of resolution, 249
max_locals item
  (Code_attribute structure), 77
memory
  exceptions
    OutOfMemoryError, 255
memory
  runtime data areas
    heap, 14
    Java virtual machine stack, 13
    layout not specified by Java virtual ma-
      chine specification, 8
  method area, 15
  native method stacks, 16
  pc register, 13
  runtime constant pool, 15
method area
definition, 15
method_info structure
  (methods table of ClassFile
    structure), 68
methods
  See also fields
  abrupt completion, 20
  area
    definition, 15
    runtime constant pool allocation from, 16
  class
    invoking, invokestatic, 374
  <clinit> method
    as class or interface initialization
methods table • monitorenter instruction

method, 24
constant_pool reference to, 58
invocation of, static constraints, 117, 120
method_info structure access flags
    ignored, 70
name_index item (method_info structure) reference, 70
code
    location, 76
    size limitation, 235
    verification, Pass 3 - bytecode verifier, 227
compilation examples, 451
constant pool references, verification process, 123
constant_pool reference to, 58
defineClass method, ClassLoader class, 243
derivation of symbolic references to at runtime, 238
descriptor
    argument number limitation, 235
    syntax and meaning, 50
    as value of CONSTANT_Utf8_info structure referenced by
descriptor_index item,
    CONSTANT_NameAndType_info structure, 63
    as value of CONSTANT_Utf8_info structure referenced by
method_descriptor_index item,
    CONSTANT_NameAndType_info structure, 93
<init> method
    invocation of, static constraints, 117
    invocation of, structural constraints, 119
    name_index item (method_info), 70
initialization, 24
instance
    data-flow analysis during class file verification, 231
    invoking, invokespecial, 370
    invoking, invokevirtual, 377
interface
    invoking, invokeinterface, 366
invocation
    instruction summary, 36
    structural constraints on instructions, 119
loadClass method, ClassLoader class, 243
lookup
during resolution, 251
dynamic, invokeinterface, 366
dynamic, invokevirtual, 377
main method, invocation of, 240
native
    pc register state during invocation, 13
    stacks, 16
normal completion, 20
number and size limitation, 235
operand stack use by, 18
protected, structural constraints, 120
requirements for throwing exceptions, 79
return
double value from, dreturn, 301
float value from, freturn, 331
instruction summary, 36
int value from, ireturn, 382
long value from, lreturn, 414
reference value from, areturn, 269
type, structural constraints on instructions, 120
void from, return, 440
synchronization, instruction summary, 37
synchronized methods
double value return from, dreturn, 301
float value return from, freturn, 331
int value return from, ireturn, 382
long value return from, lreturn, 414
reference value return from, areturn, 269
void return from, return, 440

methods table
(ClassFile structure), 47
Methods, generic, 52
methods_count item
(ClassFile structure), 46
minor_version item
(ClassFile structure), 43
monitor
    enter, monitorenter, 422
    exit, monitorexit, 424
    structured use of, 37
monitorenter instruction
    compilation examples,
synchronization, 483
monitorexit instruction

compilation examples, synchronization, 483
definition, 424

multianewarray instruction

compilation examples, arrays, 470
constraints, static, 118
definition, 426

multiplying
double, dmul, 296
float, fmul, 326
int, imul, 362
long, lmul, 408

must
instruction description implications, 257

N

name_and_type item
(CONSTANT_Fieldref_info structure), 58
(CONSTANT_Interface_Methodref_info structure), 58
(CONSTANT_Methodref_info structure), 58

name_index item
(CONSTANT_Class_info structure), 57
(CONSTANT_NameAndType_info structure), 62
(field_info structure), 67
(LocalVariableTable_attribute structure), 100, 102
(method_info structure), 70

names
See also identifiers
attributes, avoiding conflicts in, 74
classes, internal representation, 48
new attributes, 72

NaN (Not-a-Number)
conversion of
bytes item, CONSTANT_Float_info structure into, 60
high_bytes and low_bytes items, CONSTANT_Double_info structure, 62
operations that produce, 32

narrowing primitive conversions
See conversions, narrowing primitive

native method stack
definition, 16

native methods
binding, 255
invoking
class, invokeinterface, 366
class, invokevirtual, 377
instance, invokespecial, 370
instance, invokestatic, 374
pc register state during invocation, 13

negating
double, dneg, 298
float, fneg, 328
int, ineg, 363
long, lneg, 409

NegativeArraySizeException
thrown by
anewarray, 268
multianewarray, 427
newarray, 431

new instruction
compilation examples
arrays, 469
throwing exceptions, 474
working with class instances, 466
constraints, static, 118
data-flow analysis during class file verification, 231
definition, 428

newarray instruction
compilation examples, arrays, 469
constraints, static, 118
definition, 430

NoClassDefFoundError
thrown during class or interface loading, 242, 245

nonterminal symbols
descriptor grammar notation, 49

nop instruction
definition, 432

normal completion
method invocation, 20

NoSuchFieldError
thrown during field resolution, 251

NoSuchMethodError
thrown during method resolution, 252

notation
class file format descriptions, 41
field and method descriptor grammar, 49
instruction families, 31

**null reference**
definition, 12
pushing null reference, `aconst_null`, 265
testing for, 36

**Null_variable_info structure**, 87

**NullPointerException**
thrown by
  `aaload`, 262
  `aastore`, 264
  `arraylength`, 270
  `athrow`, 273
  `baload`, 275
  `bastore`, 276
  `caload`, 278
  `castore`, 279
  `daload`, 287
  `dastore`, 288
  `faload`, 317
  `fastore`, 318
  `getfield`, 336
  `iaload`, 347
  `iastore`, 349
  `invokeinterface`, 368
  `invokespecial`, 373, 379
  `laload`, 396
  `lastore`, 398
  `monitorenter`, 422
  `monitorexit`, 424
  `putfield`, 436
  `saload`, 441
  `sastore`, 442

**number_of_classes item**
  (`InnerClasses_attribute` structure), 90

**number_of_exceptions item**
  (`Exceptions_attribute` structure), 89

**numeric**
comparisons, implications of unordered NaN values, 11
conversions
  narrowing impact on precision, 34
  narrowing, support for, 34
  widening, impact on precision, 33
types
  components, 8

**O**

**Object_variable_info structure**, 88

**opcodes**
definition, 26
mnemonics by opcode (table), 487
reserved, 258

**operand stack**
data-flow analysis, 226
definition, 18
duplicating value(s)
  `dup`, 305
  `dup_x1`, 306
  `dup_x2`, 307
  `dup2`, 308
  `dup2_x1`, 309
  `dup2_x2`, 310
frames used to hold, 18
management instruction summary, 35
merging, during data-flow analysis, 229
pop value(s)
  `pop`, 433
  `pop2`, 434
size limitation, 235
structural constraints on instructions, 119
swap values, `swap`, 444

**operand(s)**
constraints, static, 116
definition, 26
implicit, compilation advantage of, 452
Java virtual machine instructions, storage order and alignment, 27
types, how distinguished by Java virtual machine instruction set, 8

**optimization**
linking
  initialization phase of, 255

**ordered values**
NaN values not ordered, implications of, 11

**ORing**

- int
  bitwise, exclusive, `ixor`, 389
  bitwise, inclusive, `ior`, 380

- long
  bitwise, exclusive, `lxor`, 421
  bitwise, inclusive, `lor`, 412
outer_class_info_index item
(classes array of
   InnerClasses_attribute
structure), 91

OutOfMemoryError
heap-related error, 15
as Java virtual machine error, 259
Java virtual machine stack-related error, 14
method area-related error, 15
native method stack-related error, 17
runtime constant pool-related error, 16

overflow
floating-point, Java virtual machine
   handling, 32
heap, 15
integer data types, not detected by Java vir-
tual machine, 32
Java virtual machine stack, 14
method area, 15
native method stack, 17
runtime constant pool, 16

overriding
   ACC_FINAL flag, method_info structure
   prevention of, 69

P
packages
   package private access, 253
   package-info file, 482
   runtime package, 241
parameterized types, 52
parameters
   descriptor, syntax and meaning, 50
pc (program counter) register
definition, 13
performance
   implications, opcode design and
   alignment, 26
polling
   for asynchronous exceptions, 25
pop instruction
definition, 433
pop2 instruction
definition, 434
popping
   operand stack value(s)
   pop, 433
pop2, 434
pound sign (#)
   use in compilation example, 450
precision
   See also numeric
   narrowing numeric conversion impact
   on, 34
   widening numeric conversion impact
   on, 33
preparation
   as part of linking, 247
primitive
   See also conversions; floating-point; inte-
gers
types
   definition, 8
   as Java virtual machine data type, 7
   values, 8
private modifier
   enforcement, 252
   methods
   invoking, invokespecial, 370
program counter
   See pc (program counter) register
protected modifier
   enforcement, 252
   fields, structural constraints, 120
   methods, structural constraints, 120
public modifier
   enforcement, 252
pushing
   byte, bipush, 277
   constants
   ldc, 401
   wide index, ldc_w, 402
double
   dconst_<d>, 291
   wide index, ldc2_w, 404
float, fconst_<f>, 321
int, icst_<i>, 350
long
   constants lconst_<l>, 400
   wide index, ldc2_w, 404
null
   object references, aconst_null, 265
short, sipush, 443
putfield instruction
   compilation examples
   operand stack operations, 473
   working with class instances, 467
putstatic instruction • ret instruction

constraints
static, 117
structural, 120, 121
definition, 435

putstatic instruction
constraints
static, 117
structural, 121
definition, 437

development

R

reference type
branch if reference comparison succeeds,
if_acmp<cond>, 352
is null, ifnonnull, 357
not null, ifnonnull, 357
determining if an object is a particular instanceof, 364
Java virtual machine
handling of, 8
data type, 7
null, testing for, 36
values
components and, 12
reference(s)
field, resolution of, 250
symbolic, mapping to concrete values as part of resolution, 249
reflection, 52
reflection
as reason for initialization, 253
Java virtual machine support for, 38
register
program counter (pc), 13
remainder
double, drem, 299
float, frem, 329
int, irem, 381
long, lrem, 413
representation
internal, class names, 48
reserved opcodes
breakpoint, 258
impdep1, 258
impdep2, 258

resolution
as part of linking, 249
errors
AbstractMethodError, thrown during method resolution, 252
ClassCircularityError, thrown during class or interface resolution, 246
IllegalAccessError, thrown during class or interface resolution, 250
IllegalAccessError, thrown during field resolution, 251
IllegalAccessError, thrown during method resolution, 252
IncompatibleClassChangeError, thrown during class or interface resolution, 246
IncompatibleClassChangeError, thrown during interface method resolution, 252
IncompatibleClassChangeError, thrown during method resolution, 251
NoSuchFieldError, thrown during field resolution, 251
NoSuchFieldError, thrown during interface method resolution, 252
NoSuchFieldError, thrown during method resolution, 251
field, 250
instructions causing anewarray, 268
checkcast, 280
getfield, 335
getstatic, 337
instanceof, 364
invokeinterface, 366
invokespecial, 370
invokestatic, 374
invokevirtual, 377
multianewarray, 426
new, 428
putfield, 435
putstatic, 437
method, instance or class, 251
method, interface, 252
ret instruction
See also jsr instruction; jsr_w instruction
compilation examples, compiling finally, 479, 481

constraints
  static, 118
  structural, 121
definition, 439
returnAddress type used by, 12
try-finally clause implementation use, Sun’s Java compiler output characteristics, 233

return
descriptor, syntax and meaning, 51
from method
double value, dreturn, 301
float value, freturn, 331
int value, ireturn, 382
long value, lreturn, 414
void, return, 440
from subroutine, ret, 439
reference value, areturn, 269
type, method, structural constraints on instructions, 120

return instruction
compilation examples
  arrays, 469, 470
  catching exceptions, 475, 476, 477, 478
  compiling finally, 479, 481
  constants and local variables in a for loop, 451, 453, 455
  while loop, 459, 460
  working with class instances, 466, 467
  throwing exceptions, 474, 475
constraints, structural, 120
definition, 440

returnAddress type
characteristics and values, 12
definition, 8
instance constraints, 121
local variable constraints, 121
right parentheses )
meanings in method descriptor, 51

round to nearest
See also numeric
definition, 32
round towards zero
definition, 32

runtime
runtime package
definition, 241
definition in Prolog, 137

runtime
class files verification issues, 122
data areas
  heap, 14
  Java virtual machine stack, 13
  method area, 15
  native method stacks, 16
  pc register, 13
  runtime constant pool, 16
RuntimeInvisibleAnnotations_attribute structure
  (attributes table of ClassFile, field_info, method_info structures), 109
RuntimeInvisibleParameterAnnotations_attribute structure
  (attributes table of method_info structure), 112
RuntimeVisibleAnnotations_attribute structure
  (attributes table of ClassFile, field_info, method_info structures), 104
RuntimeVisibleParameterAnnotations_attribute structure
  (attributes table of method_info structure), 110

S

S character
  meanings in field or method descriptor, 50

saload instruction
definition, 441

sastore instruction
definition, 442

security
  See also access_flags item
  verification of class files, 122

semantics
attributes, optional, 70
integer and floating-point operator support, 32
invokespecial instruction, access flag use to select alternatives, 45
Java virtual machine, strategies for implementing, 39
types that have no direct integer arithmetic support, 31

shift
left int, ishl, 383
left long, lshl, 415
right int
arithmetic, ishr, 384
logical, iushr, 388
right long
arithmetic, lshr, 416
logical, lushr, 420

short type
converting int to, i2s, 345
definition, 8
instruction set handling, 28
integer arithmetic not directly supported, 31
loading from arrays, saload, 441
pushing, sipush, 443
storing into arrays, sastore, 442
value range, 9

signature_index item
(Signature_attribute structure), 95

Signature_attribute structure
(attributes table of ClassFile, field_info, method_info structures), 94

Signatures, 52
signatures
characteristics and use, 48

sipush instruction
definition, 443

size
operand stacks, 19

slashes
class name use, 48

SourceDebugExtension_attribute structure
(attributes table of ClassFile structure), 97

SourceFile_attribute structure
(attributes table of ClassFile structure), 96

sourcefile_index item
(SourceFile_attribute structure), 96

SourceFile_attribute structure
(attributes table of ClassFile structure), 96

Stack map frame
See also type state
definition, 80
definition in Prolog, 127
frame types
append_frame, 83
chop_frame, 83
full_frame, 84
same_frame, 82
same_frame_extended, 83
same_locals_1_stack_item_frame, 82
same_locals_1_stack_item_frame_extended, 83
initial frame in method, 82
relationship to type state, 80

stack_map_frame structure
(StackMapTable_attribute structure), 81

StackMapTable_attribute structure
(attributes table of Code attribute), 80

StackOverflowError
definition, 259
as Java virtual machine stack-related error, 14
as native method stack-related error, 17

stacks
errors
OutOfMemoryError, 17
StackOverflowError, 14, 17
Java, 13
Java virtual machine
frames allocated from, 17
native method, 16
operand
data-flow analysis, 226
duplicating value(s), dup2, 308
duplicating value(s), dup2_x1, 309
duplicating value(s), dup2_x2, 310
duplicating value, dup, 305
duplicating value, dup_x1, 306
duplicating value, dup_x2, 307
management instruction summary, 35
maximum depth, 77
merging, during data-flow analysis, 229
pop value(s), pop2, 434
pop value, pop, 433
size limitation, 235
structural constraints on instructions, 119
swap values, swap, 444
standards
IEEE 754
adding double, conformance, dadd, 285
adding float, conformance, fadd, 315
comparing double, conformance, 
dcmp<op>, 289
comparing float, conformance, 
fcmp<op>, 319
dividing double, conformance, ddiv, 292
dividing float, conformance, fdiv, 322
floating-point comparison,
conformance, 33, 36
floating-point double format bit layout,
high_bytes and low_bytes items, 62
floating-point operation conformance to, 32
multiplying double, conformance, 
dmul, 296
multiplying float, conformance, 
fmul, 326
remainder, drem not the same as, 
drem, 299
remainder, frem not the same as, 
frem, 329
subtracting double, conformance, 
dsub, 304
subtracting float, conformance, 
fsub, 334
UTF-8 format, bibliographic reference, 65
start_pc item
(exception_table array of
Code_attribute structure), 78
(line_number_table array of
LineNumberTable_attribute structure), 98
(local_variable_table array of
LocalVariableTable_attribute structure), 100, 102
startup
Java virtual machine, 240
static
See also ACC_STATIC modifier
fields
get from classes, getstatic, 337
put into classes, putstatic, 437
methods
invoking, invokevirtual, 374
storage
automatic management system, garbage collection as, 14
data, frame use for, 17
frame allocation, 17
runtime data areas
heap, 14
Java virtual machine stack, 13
method area, 15
native method stacks, 16
pc register, 13
runtime constant pool, 15
storing
into arrays of type
byte or boolean, bastore, 276
cchar, castore, 279
double, dstore, 288
float, fstore, 318
int, istore, 349
long, lstore, 396
reference, astore, 263
short, astore, 442
into local variables of type
double, dstore, 302
double, dstore_<n>, 303
float, fstore, 332
float, fstore_<n>, 333
int, istore, 385
int, istore_<n>, 386
long, lstore, 417
long, lstore_<n>, 418
reference, astore, 271
reference, astore_<n>, 272
string_index item
(CONSTANT_String_info structure), 59
structured locking, 37
structures
class file structures, 41
subroutine
jump to
jsr, 390
wide index, jsr_w, 391
return from, ret, 439
subtracting
double, dsub, 304
float, fsub, 334
int, isub, 387
long, lsub, 419
super_class item
(ClassFile structure), 45
superclass, direct, in signatures, 52
superclasses
See also ACC_SUPER flag
checking for, 123
superinterface(s)
direct
in signatures, 52
swap instruction
definition, 444
swapping
operand stack values, swap, 444
swap instruction, operand stack manipulation
constraints, 19
symbolic references
deriving from class or interface
representation, 238
resolving, 249
synchronization
initialization implications of, 254
synchronization
compilation examples, 482
synchronized method
double value return from, dreturn, 301
float value return from, freturn, 331
int value return from, ireturn, 382
long value return from, lreturn, 414
reference value return from,
areturn, 269
void value return from, return, 440
syntax
class file specification, 42
field and method descriptor grammar, 49
internal form of class and interface
names, 48
Synthetic_attribute structure
(attributes table of ClassFile,
field_info, method_info
structures), 94

T

Table 4.8, 107
tables
in class file specification, 41
handle, 21
heap, 14
initial class, 240
initialization, 253
item, 41
Java virtual machine stack, 13
JIT (just-in-time) code generation, 449
linking, 247
loading, class or interface, 240
local variable, 18
meaning of ‘must’ in instruction
descriptions, 257
method
area, 15
current, 17
main, 240
native method stack, 16
normal completion, 20
null reference, 12
numeric
types, 8
object, 8
opcode, 27
operand, 26
stack, 18
pc register, 13
preparation, 247
primitive
types, 8
values, 8
reference
type, 12
resolution, 249
returnAddress type, 8
round to nearest, 32
round towards zero, 32
runtime constant pool, 15
signature, 48
symbolic reference, 237
user-defined class loader, 240, 242
version skew, 122
terminal symbols
descriptors grammar notation, 49
the, 107
this_class item
(ClassFile structure), 45
threads
initialization implications of multiple, 254

threads
(chapter), 485
frames use with, 17
Java virtual machine stack, 13
native method stacks, 16
pc register, 13
shared
data areas, heap, 14
data areas, method area, 15
throwing
exceptions, athrow, 273
Top_variable_info structure, 86
try-catch-finally statement
as exception handling statement, 478
try-finally statement
Sun’s Java compiler output
characteristics, 232
type parameters
formal, 52
Type state
definition in Prolog, 127
Initial in a method, 140

U

u1
as class file data type, 41
u2
as class file data type, 41
u4
as class file data type, 41
underflow
floating-point, Java virtual machine
handling, 32
integer data types, not signaled by Java vir-
tual machine, 32
Uninitialized_variable_info
structure, 88
UninitializedThis_variable_info
structure, 88
UnknownError
as Java virtual machine error, 259
 UnsatisfiedLinkError
thrown by
invokeinterface, 368
invokespecial, 373
invokestatic, 376
invokevirtual, 379
UTF-8 format
See also CONSTANT_Utf8_info structure
bibliographic reference, 65
standard, differences between Java virtual machine UTF-8 strings and, 65

V

V character
meaning in method descriptor, 51
value set conversion
definition, 22
values
concrete, mapping symbolic references to, as part of resolution, 249
return, frame use for, 17
variables
local
accessing, structural constraints on instructions, 119
code verification, Pass 3 - bytecode verifier, 227
exception handling impact on, 26
extend index by additional bytes, 
wide, 447
frames used to hold, 17
instruction specialized to handle, advantages of, 452
instructions for accessing more, summary, 31
load and store instructions, summary, 30
loading double from, dload, 294
loading double from, dload_<n>, 295
loading float from, fload, 324
loading float from, fload_<n>, 325
loading int from, iload, 360
loading int from, iload_<n>, 361
loading long from, iload, 406
loading long from, iload_<n>, 407
loading reference from, aload, 266
loading reference from, aload_<n>, 267
maximum number, 77
number limitation, 234
reuse, advantages of, 452
states, merging, during data-flow analysis, 229
storing double into, dstore, 302
storing double into, dstore_<n>, 303
storing float into, fstore, 332
storing float into, fstore_<n>, 333
storing int into, astore, 385
storing int into, astore_<n>, 386
storing long into, astore, 417
storing long into, astore_<n>, 418
storing reference into, astore, 271
storing reference into, astore_<n>, 272

verification
as part of linking, 247
by type checking, 123
by type inference, 226
classIsTypeSafe predicate, 125
errors
VERIFYError, thrown during class or interface verification, 247
subtyping, 131
type hierarchy, 128
verification_type_info structure, 86
verify
thrown during class or interface linking, 247
versions
binary compatibility issues, 122
major, major_version item (ClassFile structure) representation of, 43
minor, minor_version item (ClassFile structure) representation of, 43
VirtualMachineError
definition, 258
void
field descriptor specification, 51
returning from method invocation, return, 440

W

while keyword
compilation examples, 458
wide instruction
constraints, static, 116
definition, 447
Z

Z character
  meaning in field or method descriptor, 50