The Java® Virtual Machine Specification
Java SE 7 Edition
This page intentionally left blank
# Table of Contents

Preface to the Java SE 7 Edition  xiii

Preface to the Second Edition  xv

Preface to the First Edition  xvii

## 1 Introduction  1

1.1 A Bit of History  1
1.2 The Java Virtual Machine  2
1.3 Summary of Chapters  3
1.4 Notation  4

## 2 The Structure of the Java Virtual Machine  5

2.1 The `class` File Format  5
2.2 Data Types  6
2.3 Primitive Types and Values  6
   2.3.1 Integral Types and Values  7
   2.3.2 Floating-Point Types, Value Sets, and Values  8
   2.3.3 The `returnAddress` Type and Values  10
   2.3.4 The `boolean` Type  10
2.4 Reference Types and Values  11
2.5 Run-Time Data Areas  11
   2.5.1 The `pc` Register  12
   2.5.2 Java Virtual Machine Stacks  12
   2.5.3 Heap  13
   2.5.4 Method Area  13
   2.5.5 Run-Time Constant Pool  14
   2.5.6 Native Method Stacks  14
2.6 Frames  15
   2.6.1 Local Variables  16
   2.6.2 Operand Stacks  17
   2.6.3 Dynamic Linking  18
   2.6.4 Normal Method Invocation Completion  18
   2.6.5 Abrupt Method Invocation Completion  18
2.7 Representation of Objects  19
2.8 Floating-Point Arithmetic  19
   2.8.1 Java Virtual Machine Floating-Point Arithmetic and IEEE 754  19
   2.8.2 Floating-Point Modes  20
2.8.3 Value Set Conversion 20
2.9 Special Methods 22
2.10 Exceptions 23
2.11 Instruction Set Summary 25
2.11.1 Types and the Java Virtual Machine 26
2.11.2 Load and Store Instructions 29
2.11.3 Arithmetic Instructions 30
2.11.4 Type Conversion Instructions 32
2.11.5 Object Creation and Manipulation 34
2.11.6 Operand Stack Management Instructions 34
2.11.7 Control Transfer Instructions 34
2.11.8 Method Invocation and Return Instructions 35
2.11.9 Throwing Exceptions 36
2.11.10 Synchronization 36
2.12 Class Libraries 37
2.13 Public Design, Private Implementation 37

3 Compiling for the Java Virtual Machine 39
3.1 Format of Examples 39
3.2 Use of Constants, Local Variables, and Control Constructs 40
3.3 Arithmetic 45
3.4 Accessing the Run-Time Constant Pool 46
3.5 More Control Examples 47
3.6 Receiving Arguments 49
3.7 Invoking Methods 50
3.8 Working with Class Instances 53
3.9 Arrays 55
3.10 Compiling Switches 57
3.11 Operations on the Operand Stack 58
3.12 Throwing and Handling Exceptions 59
3.13 Compiling finally 63
3.14 Synchronization 66
3.15 Annotations 67

4 The class File Format 69
4.1 The ClassFile Structure 70
4.2 The Internal Form of Names 75
4.2.1 Binary Class and Interface Names 75
4.2.2 Unqualified Names 75
4.3 Descriptors and Signatures 76
4.3.1 Grammar Notation 76
4.3.2 Field Descriptors 77
4.3.3 Method Descriptors 78
4.3.4 Signatures 79
4.4 The Constant Pool 82
4.4.1 The CONSTANT_Class_info Structure 83
4.4.2 The CONSTANT_Fieldref_info, CONSTANT_Methodref_info, and CONSTANT_InterfaceMethodref_info Structures 84
4.4.3 The CONSTANT_String_info Structure 86
4.4.4 The CONSTANT_Integer_info and CONSTANT_Float_info Structures 86
4.4.5 The CONSTANT_Long_info and CONSTANT_Double_info Structures 88
4.4.6 The CONSTANT_NameAndType_info Structure 89
4.4.7 The CONSTANT_Utf8_info Structure 90
4.4.8 The CONSTANT_MethodHandle_info Structure 92
4.4.9 The CONSTANT_MethodType_info Structure 93
4.4.10 The CONSTANT_InvokeDynamic_info Structure 94

4.5 Fields 95
4.6 Methods 97
4.7 Attributes 100
4.7.1 Defining and Naming New Attributes 102
4.7.2 The ConstantValue Attribute 103
4.7.3 The Code Attribute 104
4.7.4 The StackMapTable Attribute 107
4.7.5 The Exceptions Attribute 115
4.7.6 The InnerClasses Attribute 116
4.7.7 The EnclosingMethod Attribute 119
4.7.8 The Synthetic Attribute 120
4.7.9 The Signature Attribute 120
4.7.10 The SourceFile Attribute 121
4.7.11 The SourceDebugExtension Attribute 122
4.7.12 The LineNumberTable Attribute 123
4.7.13 The LocalVariableTable Attribute 124
4.7.14 The LocalVariableTypeTable Attribute 126
4.7.15 The Deprecated Attribute 128
4.7.16 The RuntimeVisibleAnnotations attribute 128
4.7.16.1 The element_value structure 130
4.7.17 The RuntimeInvisibleAnnotations attribute 133
4.7.18 The RuntimeVisibleParameterAnnotations attribute 134
4.7.19 The RuntimeInvisibleParameterAnnotations attribute 136
4.7.20 The AnnotationDefault attribute 137
4.7.21 The BootstrapMethods attribute 138

4.8 Format Checking 140
4.9 Constraints on Java Virtual Machine code 140
4.9.1 Static Constraints 141
4.9.2 Structural Constraints 144

4.10 Verification of class Files 148
4.10.1 Verification by Type Checking 149
4.10.1.1 Accessors for Java Virtual Machine Artifacts 152
4.10.1.2 Verification Type System 155
4.10.1.3 Instruction Representation 159
4.10.1.4 Stack Map Frame Representation 160
4.10.1.5 Type Checking Abstract and Native Methods 166
The Java® Virtual Machine Specification

4.10.1.6 Type Checking Methods with Code 167
4.10.1.7 Type Checking Load and Store Instructions 174
4.10.1.8 Type Checking for protected Members 176
4.10.1.9 Type Checking Instructions 179
4.10.2 Verification by Type Inference 327
4.10.2.1 The Process of Verification by Type Inference 327
4.10.2.2 The Bytecode Verifier 328
4.10.2.3 Values of Types long and double 330
4.10.2.4 Instance Initialization Methods and Newly Created Objects 331
4.10.2.5 Exceptions and finally 332
4.11 Limitations of the Java Virtual Machine 334

5 Loading, Linking, and Initializing 337
5.1 The Run-Time Constant Pool 337
5.2 Java Virtual Machine Startup 340
5.3 Creation and Loading 340
5.3.1 Loading Using the Bootstrap Class Loader 342
5.3.2 Loading Using a User-defined Class Loader 343
5.3.3 Creating Array Classes 344
5.3.4 Loading Constraints 344
5.3.5 Deriving a Class from a class File Representation 346
5.4 Linking 347
5.4.1 Verification 348
5.4.2 Preparation 348
5.4.3 Resolution 349
5.4.3.1 Class and Interface Resolution 350
5.4.3.2 Field Resolution 351
5.4.3.3 Method Resolution 352
5.4.3.4 Interface Method Resolution 353
5.4.3.5 Method Type and Method Handle Resolution 354
5.4.3.6 Call Site Specifier Resolution 357
5.4.4 Access Control 358
5.4.5 Method overriding 359
5.5 Initialization 359
5.6 Binding Native Method Implementations 362
5.7 Java Virtual Machine Exit 362

6 The Java Virtual Machine Instruction Set 363
6.1 Assumptions: The Meaning of "Must" 363
6.2 Reserved Opcodes 364
6.3 Virtual Machine Errors 364
6.4 Format of Instruction Descriptions 365
6.5 Instructions 368
   aaload 369
   aastore 370
aconst_null 372
aload 373
aload_<n> 374
anewarray 375
areturn 376
arraylength 377
astore 378
astore_<n> 379
athrow 380
baload 382
bastore 383
bipush 384
caload 385
castore 386
checkcast 387
d2f 389
d2i 390
d2l 391
dadd 392
daload 394
dastore 395
dcmp<op> 396
dconst_<d> 398
ddiv 399
dload 401
dload_<n> 402
dmul 403
dneg 405
drem 406
dreturn 408
dstore 409
dstore_<n> 410
dsub 411
dup 412
dup_x1 413
dup_x2 414
dup2 415
dup2_x1 416
dup2_x2 417
f2d 419
f2i 420
f2l 421
fadd 422
faload 424
fastore 425
fcmp<op> 426
fconst_<f> 428
fdiv 429
The Java Virtual Machine Specification

fload 431
fload_<n> 432
fmul 433
fneg 435
frem 436
freturn 438
fstore 439
fstore_<n> 440
fsub 441
gffield 442
gestatic 444
goto 446
goto_w 447
i2b 448
i2c 449
i2d 450
i2f 451
i2l 452
i2s 453
iadd 454
iaload 455
iand 456
iastore 457
iconst_<i> 458
idiv 459
if_acmp<cond> 460
if_icmp<cond> 461
if<cond> 463
ifnonnull 465
ifnull 466
iinc 467
iload 468
iload_<n> 469
imul 470
ineg 471
instanceof 472
invokedynamic 474
invokeinterface 479
invokespecial 482
invokestatic 486
invokevirtual 489
ior 494
irem 495
ireturn 496
ishl 497
ishr 498
istore 499
istore_<n> 500
isub 501
iushr 502
ixor 503
jsr 504
jsr_w 505
l2d 506
l2f 507
l2i 508
ladd 509
laload 510
land 511
lastore 512
lcmp 513
lconst_1 514
ldc 515
ldc_w 517
ldc2_w 519
ldiv 520
lload 521
lload_1 522
lmul 523
lneg 524
lookupswitch 525
lor 527
lrem 528
lreturn 529
lshl 530
lshr 531
lstore 532
lstore_1 533
lsub 534
lushr 535
lxor 536
monitorenter 537
monitorexit 539
multianewarray 541
new 543
newarray 545
nop 547
pop 548
pop2 549
putfield 550
putstatic 552
ret 554
return 555
saload 556
sastore 557
sipush 558
7 Opcode Mnemonics by Opcode 565

Index 569

A Limited License Grant 587
Preface to the Java SE 7 Edition

The Java® SE 7 Edition of The Java Virtual Machine Specification incorporates all the changes that have been made to the Java Virtual Machine since the Second Edition in 1999. In addition, numerous corrections and clarifications have been made to align with popular implementations of the Java Virtual Machine, and with concepts common to the Java Virtual Machine and the Java programming language.

Readers may send feedback about errors and ambiguities in The Java Virtual Machine Specification to jvms-comments_ww@oracle.com.

The Java SE 5.0 platform in 2004 brought momentous changes to the Java programming language but had a relatively muted effect on the design of the Java Virtual Machine. Additions were made to the class file format to support new Java programming language features such as generics and variable arity methods.

The Java SE 6 platform in 2006 saw no changes to the Java programming language but an entirely new approach to bytecode verification in the Java Virtual Machine. Eva Rose, in her Master's Thesis, proposed a radical revision of bytecode verification in the context of the Java Card platform. This led to an implementation for Java ME CLDC, and eventually to the revision of the Java SE verification process documented in Chapter 4.

Sheng Liang implemented the Java ME CLDC verifier. Antero Taivalsaari led the overall specification of Java ME CLDC and Gilad Bracha was responsible for specifying the verifier. Alessandro Coglio's analysis of bytecode verification was the most extensive, realistic, and thorough study of the topic, and contributed greatly to the specification. Wei Tao, together with Frank Yellin, Tim Lindholm, and Gilad Bracha, 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 HotSpot JVM. Later, Mingyao Yang improved the design and specification, and implemented the final version that shipped in the Reference Implementation of Java SE 6. The specification also 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.

The Java SE 7 platform in 2011 made good on the promise given in the First Edition of The Java Virtual Machine Specification in 1997: "In the future, we will consider bounded extensions to the Java virtual machine to provide better support for other languages." Gilad Bracha, in his work on hotswapping, anticipated the burden of
the Java Virtual Machine's static type system on implementers of dynamically-typed languages. Consequently, the *invokedynamic* instruction and its supporting infrastructure were developed by John Rose and the JSR 292 Expert Group: Ola Bini, Rémi Forax, Dan Heidinga, Fredrik Öhrström, and Jochen Theodorou, with special contributions from Charlie Nutter and Christian Thalinger.

More people than we can mention here have, over time, contributed to the design and implementation of the Java Virtual Machine. The excellent performance we see in the Java Virtual Machine implementations 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 the HotSpot JVM. Lars Bak and Urs Hölzle 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 Java Virtual Machine implementations today.

This specification has been significantly improved thanks to contributions from Martin Buchholz, Brian Goetz, Paul Hohensee, David Holmes, Karen Kinnear, Keith McGuigan, Jeff Nisewanger, Mark Reinhold, Naoto Sato, and Bill Pugh, as well as Uday Dhanikonda, Janet Koenig, Adam Messinger, John Pampuch, Georges Saab, and Bernard Traversat. Jon Courtney and Roger Riggs helped to ensure this specification is applicable to Java ME as much as Java SE. Leonid Arbouzov, Stanislav Avzan, Yuri Gaevsky, Ilya Mukhin, Sergey Reznick, and Kirill Shirokov have done outstanding work in the Java Compatibility Kit to ensure this specification is both testable and tested.

Gilad Bracha  
*Los Altos, California*

Alex Buckley  
*Santa Clara, California*

*June, 2011*
Preface to the Second Edition

This 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 Editions 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
Palo Alto, California

Frank Yellin
Redwood City, California

April, 1999
This page intentionally left blank
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.

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 any specific implementation. This specification 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 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.

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 the Java Software group at Sun.

This book began as internal project documentation edited by Kathy Walrath. 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. 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.

Tim Lindholm
Palo Alto, California

Frank Yellin
Redwood City, California

June, 1996
This page intentionally left blank
CHAPTER 2

The Structure of the Java Virtual Machine

This document specifies an abstract machine. It does not describe any particular implementation of the Java Virtual Machine.

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 run-time 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.

All references to Unicode in this specification are given with respect to The Unicode Standard, Version 6.0.0, available at http://www.unicode.org/.

2.1 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.
2.2 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 run time, 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 run time, 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, `iadd`, `ladd`, `fadd`, and `dadd` are all Java Virtual Machine instructions that add two numeric values and produce numeric results, but each is specialized for its operand type: `int`, `long`, `float`, and `double`, respectively. For a summary of type support in the Java Virtual Machine instruction set, see §2.11.1.

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 *reference*. Values of type *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 *reference*.

2.3 Primitive Types and Values

The primitive data types supported by the Java Virtual Machine are the *numeric types*, the *boolean type* (§2.3.4), and the *returnAddress type* (§2.3.3).

The numeric types consist of the *integral types* (§2.3.1) and the *floating-point types* (§2.3.2).

The integral types are:

- *byte*, whose values are 8-bit signed two's-complement integers, and whose default value is zero
- *short*, whose values are 16-bit signed two's-complement integers, and whose default value is zero
• **int**, whose values are 32-bit signed two's-complement integers, and whose default value is zero

• **long**, whose values are 64-bit signed two's-complement integers, and whose default value is zero

• **char**, whose values are 16-bit unsigned integers representing Unicode code points in the Basic Multilingual Plane, encoded with UTF-16, and whose default value is the null code point (‘\u0000’)

The floating-point types are:

• **float**, whose values are elements of the float value set or, where supported, the float-extended-exponent value set, and whose default value is positive zero

• **double**, whose values are elements of the double value set or, where supported, the double-extended-exponent value set, and whose default value is positive zero

The values of the **boolean** type encode the truth values **true** and **false**, and the default value is **false**.

*The Java Virtual Machine Specification, First Edition* did not consider **boolean** to be a Java Virtual Machine type. However, **boolean** values do have limited support in the Java Virtual Machine. *The Java Virtual Machine Specification, Second Edition* clarified the issue by treating **boolean** as a type.

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.

### 2.3.1 Integral Types and Values

The values of the integral types of the Java Virtual Machine are:

• 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
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, 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 $E_{\min} = -(2^{K-1}-2)$ and $E_{\max} = 2^{K-1}-1$, 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 $E_{\min}$ and $E_{\max}$) for the two required and two optional floating-point value sets are summarized in Table 2.1.
Table 2.1. 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 2.1; 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 2.1 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 can 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.

### 2.3.3 The `returnAddress` Type and Values

The `returnAddress` type is used by the Java Virtual Machine's `jsr`, `ret`, and `jsr_w` instructions (§`jsr`, §`ret`, §`jsr_w`). 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.

### 2.3.4 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 (§`newarray`) enables creation of `boolean` arrays. Arrays of type `boolean` are accessed and modified using the `byte` array instructions `baload` and `bastore` (§`baload`, §`bastore`).
In Oracle’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 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.

2.4 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.

An array type consists of a component type with a single dimension (whose length is not given by the type). The component type of an array type may itself be an array type. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; this is called the element type of the array type. The element type of an array type is necessarily either a primitive type, or a class type, or an interface type.

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 run-time type, but may be cast to any type. The default value of a reference type is null. The Java Virtual Machine specification does not mandate a concrete value encoding null.

2.5 Run-Time Data Areas

The Java Virtual Machine defines various run-time 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.
2.5.1 The pc Register

The Java Virtual Machine can support many threads of execution at once (JLS §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 (§2.6) 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.

2.5.2 Java Virtual Machine Stacks

Each Java Virtual Machine thread has a private Java Virtual Machine stack, created at the same time as the thread. A Java Virtual Machine stack stores frames (§2.6). 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.

In The Java Virtual Machine Specification, First Edition, the Java Virtual Machine stack was known as the Java stack.

This 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 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.

2.5.3 Heap

The Java Virtual Machine has a heap that is shared among all Java Virtual Machine threads. The heap is the run-time 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.

2.5.4 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 an operating system process. It stores per-class structures such as the run-time constant pool, field and method data, and the code for methods and constructors, including the special methods (§2.9) 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`.

### 2.5.5 Run-Time Constant Pool

A run-time constant pool is a per-class or per-interface run-time 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 run-time. The run-time 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 run-time constant pool is allocated from the Java Virtual Machine's method area (§2.5.4). The run-time 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 run-time constant pool for a class or interface:

- When creating a class or interface, if the construction of the run-time 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`.

See §5 for information about the construction of the run-time constant pool.

### 2.5.6 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.

This 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.

A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the native method stacks, as well as, in the case of varying-size native method stacks, 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`.

## 2.6 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 (§2.5.2) of the thread creating the frame. Each frame has its own array of local variables (§2.6.1), its own operand stack (§2.6.2), and a reference to the runtime constant pool (§2.5.5) of the class of the current method.

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

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.

### 2.6.1 Local Variables

Each frame (§2.6) 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 long and double 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.

2.6.2 Operand Stacks

Each frame (§2.6) 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 iadd instruction (§iadd) adds two int values together. It requires that the int values to be added be the top two values of the operand stack, pushed there by previous instructions. Both of the 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 long or type 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 int values and subsequently treat them as a long or to push two float values and subsequently add them with an iadd instruction. A small number of Java Virtual Machine instructions (the dup instructions (§dup) and swap (§swap)) operate on run-time 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 class file verification (§4.10).
At any point in time, an operand stack has an associated depth, where a value of type `long` or `double` contributes two units to the depth and a value of any other type contributes one unit.

### 2.6.3 Dynamic Linking

Each frame (§2.6) contains a reference to the run-time constant pool (§2.5.5) for the type of the current method to support *dynamic linking* of the method code. The 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 run-time 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.

### 2.6.4 Normal Method Invocation Completion

A method invocation *completes normally* if that invocation does not cause an exception (§2.10) 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 (§2.11.8), the choice of which must be appropriate for the type of the value being returned (if any).

The current frame (§2.6) 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.

### 2.6.5 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 (§2.10), and that exception is not handled within the method. Execution of an `athrow` instruction (§athrow) 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.

### 2.7 Representation of Objects

The Java Virtual Machine does not mandate any particular internal structure for objects.

In some of Oracle’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.

### 2.8 Floating-Point Arithmetic


#### 2.8.1 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.

2.8.2 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 (§2.3.2), 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 (§2.8.3). If a double-extended-exponent value set is supported (§2.3.2), 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.

2.8.3 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.
2.9 Special Methods

At the level of the Java Virtual Machine, every constructor written in the Java programming language (JLS §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 (§invokespecial), and they may be invoked only on uninitialized class instances. An instance initialization method takes on the access permissions (JLS §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 has the special name `<clinit>`, takes no arguments, and is void (§4.3.3).

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.

In a class file whose version number is 51.0 or above, the method must additionally have its ACC_STATIC flag (§4.6) set in order to be the class or interface initialization method.

This requirement is new in Java SE 7. In a class file whose version number is 50.0 or below, a method named `<clinit>` that is void and takes no arguments is considered the class or interface initialization method regardless of the setting of its ACC_STATIC flag.

The name `<clinit>` 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.

A method is *signature polymorphic* if and only if all of the following conditions hold:

- It is declared in the `java.lang.invoke.MethodHandle` class.
- It has a single formal parameter of type `Object[]`.
- It has a return type of `Object`.
- It has the ACC_VARARGS and ACC_NATIVE flags set.
In Java SE 7, the only signature polymorphic methods are the `invoke` and `invokeExact` methods of the class `java.lang.invoke.MethodHandle`.

The Java Virtual Machine gives special treatment to signature polymorphic methods in the `invokevirtual` instruction (§`invokevirtual`), in order to effect invocation of a `method handle`. A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation (§5.4.3.5), with optional transformations of arguments or return values. These transformations are quite general, and include such patterns as conversion, insertion, deletion, and substitution. See the `java.lang.invoke` package in the Java SE platform API for more information.

### 2.10 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:

- An `athrow` instruction (§`athrow`) was executed.
- 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:
    - 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.
- 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.

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.

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 handler 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 (§2.6.5). 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 run time, 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 (§3.12). When class files are generated by some other means, the defined search procedure ensures that all Java Virtual Machine implementations will behave consistently.

### 2.11 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

```java
    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, `byte1` and `byte2`, such that its value is `(byte1 << 8) | byte2`.

The bytecode instruction stream is only single-byte aligned. The two exceptions are the `lookupswitch` and `tableswitch` instructions (§`lookupswitch`, §`tableswitch`).
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 run time on many machines.

2.11.1 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 `iload` instruction (§\texttt{iload}) loads the contents of a local variable, which must be an \texttt{int}, onto the operand stack. The `fload` instruction (§\texttt{fload}) 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, such as `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 instruction supported all of the Java Virtual Machine's run-time 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 2.2 summarizes the type support in the instruction set of the Java Virtual Machine. A specific instruction, with type information, is built by replacing the \texttt{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 \texttt{int}, \texttt{iload}, but there is no load instruction for type \texttt{byte}.

Note that most instructions in Table 2.2 do not have forms for the integral types \texttt{byte}, \texttt{char}, and \texttt{short}. None have forms for the \texttt{boolean} type. A compiler encodes loads of literal values of types \texttt{byte} and \texttt{short} using Java Virtual Machine
instructions that sign-extend those values to values of type int at compile-time or run-time. 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 run-time. 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 2.2. 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>Tipush</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tconst</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tstore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tinc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tastore</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tsub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tdiv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tneg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tshr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tushr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tand</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Txor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmpl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tcmpg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if_TcmpOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treturn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 2.3.

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 2.3.

### Table 2.3. Actual and Computational types in the Java Virtual Machine

<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>1</td>
</tr>
<tr>
<td>byte</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>char</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>1</td>
</tr>
<tr>
<td>reference</td>
<td>reference</td>
<td>1</td>
</tr>
<tr>
<td>returnAddress</td>
<td>returnAddress</td>
<td>1</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>2</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>2</td>
</tr>
</tbody>
</table>

### 2.11.2 Load and Store Instructions

The load and store instructions transfer values between the local variables (§2.6.1) and the operand stack (§2.6.2) of a Java Virtual Machine frame (§2.6):


- Store a value from the operand stack into a local variable: *istore*, *istore_<n>*<ref>, *lstore*, *lstore_<n>*<ref>, *fstore*, *fstore_<n>*<ref>, *dstore*, *dstore_<n>*<ref>, *astore*, *astore_<n>*<ref>.

- Load a constant on to the operand stack: *bipush*, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *iconst_<i>*<ref>, *lconst_<l>*<ref>, *fconst_<f>*<ref>, *dconst_<d>*<ref>.

- 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 (§2.11.5) 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 (§2.11.1).

This notation for instruction families is used throughout this specification.

2.11.3 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 (§2.11.1), 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: \textit{iinc}.

• Comparison: \textit{dcmpg}, \textit{dcmpl}, \textit{fcmpg}, \textit{fcmpl}, \textit{lcmp}.

The semantics of the Java programming language operators on integer and floating-point values (JLS §4.2.2, JLS §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 (\textit{idiv} and \textit{ldiv}) and the integer remainder instructions (\textit{irem} and \textit{lrem}), which throw an \textit{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 \textit{denormalized} floating-point numbers and \textit{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. \textit{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 \textit{round to nearest} mode.

The Java Virtual Machine uses the IEEE 754 \textit{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 run-time 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 \texttt{long} (\textit{lcmp}) perform a signed comparison. Comparisons on values of floating-point types (\textit{dcmpg}, \textit{dcmpl}, \textit{fcmpg}, \textit{fcmpl}) are performed using IEEE 754 nonsignaling comparisons.
2.11.4 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 widening numeric conversion instructions are `i2l`, `i2f`, `i2d`, `l2f`, `l2d`, and `f2d`. 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 (§2.8.2) 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 run-time 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 §2.11.1, 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 run-time exception (not to be confused with an IEEE 754 floating-point exception).

2.11.5 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`.

2.11.6 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`.

2.11.7 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`, `ifne`, `iflt`, `ifle`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmple`, `if_icmpgt`, `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 it is not required to specify a concrete value for null (§2.4).

Conditional branches on comparisons between data of types boolean, byte, char, and short are performed using int comparison instructions (§2.11.1). 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 (§2.11.3). 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.

### 2.11.8 Method Invocation and Return Instructions

The following five 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 run-time object to find the appropriate method.
- **invokespecial** invokes an instance method requiring special handling, whether an instance initialization method (§2.9), a private method, or a superclass method.
- **invokestatic** invokes a class (static) method in a named class.
- **invokedynamic** invokes the method which is the target of the call site object bound to the invokedynamic instruction. The call site object was bound to a specific lexical occurrence of the invokedynamic instruction by the Java Virtual Machine as a result of running a bootstrap method before the first execution of the instruction. Therefore, each occurrence of an invokedynamic instruction has a unique linkage state, unlike the other instructions which invoke methods.

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

An exception is thrown programmatically using the *athrow* instruction. Exceptions can also be thrown by various Java Virtual Machine instructions if they detect an abnormal condition.

2.11.10 Synchronization

The Java Virtual Machine supports synchronization of both methods and sequences of instructions within a method by a single synchronization construct: the *monitor*. Method-level synchronization is performed implicitly, as part of method invocation and return (§2.11.8). A *synchronized* method is distinguished in the run-time constant pool's *method_info* structure (§4.6) by the ACC_SYNCHRONIZED flag, which is checked by the method invocation instructions. When invoking a method for which 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 monitorenter and monitorexit instructions to support such language constructs. Proper implementation of *synchronized* blocks requires cooperation from a compiler targeting the Java Virtual Machine (§3.14).

*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.

2.12 Class Libraries

The Java Virtual Machine must provide sufficient support for the implementation of the class libraries of the Java SE 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 specification. See the specifications of the Java SE platform class libraries for details.

2.13 Public Design, Private Implementation

Thus far this specification 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 SE 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.

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, Oracle works with other Java Virtual Machine implementors and with tool vendors to develop common interfaces to the Java Virtual Machine for use by such tools, and to promote those interfaces across the industry.

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.

- 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.
Index

A

a bit of history, 1
aaload, 181, 369
    stack map frame representation, 162
aastore, 182, 370
abrupt method invocation completion, 18
    Exceptions, 24
    invokedynamic, 477
    synchronization, 67
access control, 358
    class and interface resolution, 351
    field resolution, 351
    method resolution, 353
    type checking for protected members, 176
accessing the run-time constant pool, 46
accessors for Java Virtual Machine artifacts, 152
    type checking instructions, 179
    verification by type checking, 151
aconst_null, 183, 372
actual and computational types in the Java Virtual Machine, 29
    types and the Java Virtual Machine, 29, 29
aload, 184, 373
    astore, 378
    wide, 562
aload_<n>, 185, 374
    astore_<n>, 379
anewarray, 186, 375
    multianewarray, 542
AnnotationDefault attribute, 137
    methods, 100, 100
annotations, 67
areturn, 187, 376
arithmetic, 45
arithmetic instructions, 30
    control transfer instructions, 35
array class loading
    creation and loading, 341
    loading constraints, 345
array type codes, 545
arraylength, 188, 377
    getfield, 443
    stack map frame representation, 162
arrays, 55
assumptions: the meaning of "must", 363
astore, 189, 378
   aload, 373
    wide, 562
astore_<n>, 190, 379
    aload_<n>, 374
athrow, 191, 380
    abrupt method invocation completion, 18
    Exceptions, 23
attributes, 100
    ClassFile structure, 74, 74
    Code attribute, 107
    fields, 96, 96
    methods, 99, 99

B

baload, 192, 382
    boolean type, 10
C

call site specifier resolution, 357
  invokedynamic, 474
caload, 195, 385
castore, 196, 386
castore, 197, 387
  instanceof, 473
class access and property modifiers, 71
  ClassFile structure, 71, 72, 72
  invokespecial, 482
class and interface resolution, 350
  anewarray, 375, 375
  castore, 387, 388
  class and interface resolution, 351
    deriving a class from a class file representation, 346, 347
  field resolution, 351
  instanceof, 472, 473
  interface method resolution, 353
    ldc, 515, 516
    ldc_w, 517, 518
    method resolution, 352, 352
    method type and method handle resolution, 354, 356
    multianewarray, 541, 542
    new, 543, 543
class file format, 5, 69
  assumptions: the meaning of "must", 363
    creation and loading, 341
    reserved opcodes, 364
class libraries, 37
  attributes, 101, 101
    creation and loading, 340
    initialization, 359
class loading
  access control, 358
  class and interface resolution, 350
  creating array classes, 344
format checking, 140
getfield, 442
invokespecial, 482
invokevirtual, 489
putfield, 550
run-time constant pool, 14, 337
stack map frame representation, 161
verification, 348

**ClassFile structure, 70**
annotations, 67, 67
attributes, 100
BootstrapMethods attribute, 138
Depreciated attribute, 128
deriving a class from a class file representation, 346, 346
EnclosingMethod attribute, 119
format checking, 140
InnerClasses attribute, 116
limitations of the Java Virtual Machine, 335, 335, 335
loading using a user-defined class loader, 343
RuntimeInvisibleAnnotations attribute, 133
RuntimeVisibleAnnotations attribute, 128
Signature attribute, 120
SourceDebugExtension attribute, 122
SourceFile attribute, 121
Synthetic attribute, 120
verification by type checking, 149

**Code attribute, 104**
attributes, 100
constraints on Java Virtual Machine code, 140
Exceptions, 25
frames, 15
limitations of the Java Virtual Machine, 335, 335
LineNumberTable attribute, 123
local variables, 16
LocalVariableTable attribute, 124
LocalVariableTypeTable attribute, 126
methods, 100, 100
operand stacks, 17
StackMapTable attribute, 107, 115
throwing and handling Exceptions, 60, 63
verification by type checking, 150
verification of class files, 149

**compiling finally, 63**
astore, 378
astore_<n>, 379
Code attribute, 107
jsr, 504
jsr_w, 505
more control examples, 47
ret, 554

**compiling for the Java Virtual Machine, 39**
instance initialization methods and newly created objects, 331

**compiling switches, 57**
more control examples, 47

**constant pool, 82**
ClassFile structure, 71
getfield, 442
getstatic, 444
invokedynamic, 474, 474, 476
invokeinterface, 479
invokespecial, 482
invokestatic, 486
invokevirtual, 489
ldc, 515, 515, 515, 515
ldc_w, 519
ldc2_w, 517, 517, 517
method type and method handle resolution, 355
putfield, 550
putstatic, 552
resolution, 350
INDEX

run-time constant pool, 14, 337
constant pool tags, 83
constant pool, 83
constant value attribute types, 104
ConstantValue attribute, 104
CONSTANT_Class_info structure, 83
arrays, 56
binary class and interface names, 75
BootstrapMethods attribute, 140
ClassFile structure, 72, 72, 73
Code attribute, 107
CONSTANT_Fieldref_info,
CONSTANT_Methodref_info, and
CONSTANT_InterfaceMethodref_info
structures, 85
EnclosingMethod attribute, 119
Exceptions attribute, 116
InnerClasses attribute, 117, 117
ldc_w, 517
run-time constant pool, 338
StackMapTable attribute, 114
CONSTANT_Fieldref_info,
CONSTANT_Methodref_info, and
CONSTANT_InterfaceMethodref_info
structures, 84
CONSTANT_MethodHandle_info structure,
93, 93, 93
instruction representation, 159
run-time constant pool, 338, 338, 338
CONSTANT_Integer_info and
CONSTANT_Float_info structures, 86
BootstrapMethods attribute, 140
floating-point types, value sets, and values, 9
ldc, 516
ldc_w, 518
run-time constant pool, 340, 340
CONSTANT_InvokeDynamic_info structure,
94
BootstrapMethods attribute, 138
instruction representation, 159
run-time constant pool, 339
CONSTANT_Long_info and
CONSTANT_Double_info structures, 88
BootstrapMethods attribute, 140
ClassFile structure, 71
floating-point types, value sets, and values, 9
ldc2_w, 519
run-time constant pool, 340, 340
CONSTANT_MethodHandle_info structure,
92
BootstrapMethods attribute, 139, 139, 140
run-time constant pool, 338
CONSTANT_MethodType_info structure, 93
BootstrapMethods attribute, 140
run-time constant pool, 339
CONSTANT_NameAndType_info structure,
89
binary class and interface names, 75
CONSTANT_Fieldref_info,
CONSTANT_Methodref_info, and
CONSTANT_InterfaceMethodref_info
structures, 85
CONSTANT_InvokeDynamic_info
structure, 94
EnclosingMethod attribute, 119
run-time constant pool, 340
CONSTANT_String_info structure, 86
BootstrapMethods attribute, 140
run-time constant pool, 339
CONSTANT_Utf8_info structure, 90
AnnotationDefaultValue attribute, 138
attributes, 100
binary class and interface names, 75
BootstrapMethods attribute, 139
Code attribute, 105
CONSTANT_Class_info structure, 84
CONSTANT_MethodType_info structure, 94
CONSTANT_NameAndType_info structure, 90, 90
CONSTANT_String_info structure, 86
ConstantValue attribute, 104
Deprecated attribute, 119
descriptors and signatures, 76
element_value structure, 132, 132, 132
EnclosingMethod attribute, 119
Exceptions attribute, 115
fields, 96, 96
InnerClasses attribute, 116, 117
limitations of the Java Virtual Machine, 335
LineNumberTable attribute, 123
LocalVariableTable attribute, 124, 125, 125
LocalVariableTypeTable attribute, 126, 127, 127
methods, 99, 99
run-time constant pool, 340
RuntimeInvisibleAnnotations attribute, 134
RuntimeInvisibleParameterAnnotations attribute, 136
RuntimeVisibleAnnotations attribute, 129, 130, 130
RuntimeVisibleParameterAnnotations attribute, 135
Signature attribute, 121, 121
SourceDebugExtension attribute, 122, 123
SourceFile attribute, 121, 122
StackMapTable attribute, 108
Synthetic attribute, 120
ConstantValue attribute, 103
fields, 96, 96
initialization, 360
limitations of the Java Virtual Machine, 335
constraints
creating array classes, 344
deriving a class from a class file representation, 347
field resolution, 352
interface method resolution, 354
loading using a user-defined class loader, 343
method resolution, 353
preparation, 348
constraints on Java Virtual Machine code, 140
Code attribute, 106
verification, 348, 348
control transfer instructions, 34
creating array classes, 344
creation and loading, 341
loading constraints, 345
creation and loading, 340
access control, 358
class and interface resolution, 350
creating array classes, 344
format checking, 140
getfield, 442
invokespecial, 482
invokevirtual, 489
putfield, 550
run-time constant pool, 14, 337
stack map frame representation, 161
verification, 348

D
d2f, 198, 389
d2i, 199, 390
d2l, 200, 391
dadd, 201, 392
daload, 202, 394
dastore, 203, 395
data types, 6
baload, 382
INDEX

bastore, 383
dcmp<op>, 204, 396
dconst_<d>, 205, 398
ddiv, 206, 399

defineclass
   creation and loading, 342
   loading constraints, 345
   loading using a user-defined class loader, 343
   loading using the bootstrap class loader, 343

defining and naming new attributes, 102
   ClassFile structure, 74
   Code attribute, 107
   fields, 97
   methods, 100

Deprecated attribute, 128
   ClassFile structure, 74
   fields, 96
   methods, 100

deriving a class from a class file representation, 346
   creation and loading, 342
   loading constraints, 345
   loading using a user-defined class loader, 343
   loading using the bootstrap class loader, 343

descriptors and signatures, 76
   binary class and interface names, 75
dload, 207, 401
   wide, 562
dload_<n>, 208, 402
dmul, 209, 403
dneg, 210, 405
drem, 211, 406
dreturn, 212, 408
dstore, 213, 409
   wide, 562
dstore_<n>, 214, 410
dsub, 215, 411
dup, 216, 412

operand stacks, 17
dup2, 219, 415
dup2_x1, 220, 416
dup2_x2, 221, 417
dup_x1, 217, 413
dup_x2, 218, 414
dynamic linking, 18

E
element_value structure, 130
EnclosingMethod attribute, 119
   ClassFile structure, 74

Exceptions, 23
   abrupt method invocation completion, 18
   athrow, 380
   Code attribute, 106
   normal method invocation completion, 18
   synchronization, 67
   throwing and handling Exceptions, 60, 62
   Virtual Machine errors, 364

Exceptions and finally, 332
   compiling finally, 63
   jsr, 504
   jsr_w, 505
   ret, 554

Exceptions attribute, 115
   methods, 100, 100

F
f2d, 223, 419
f2i, 224, 420
f2l, 225, 421
fadd, 226, 422
faload, 227, 424
fastore, 228, 425
fcmp<op>, 229, 426
fconst_<f>, 230, 428
fdiv, 231, 429
field access and property flags, 95
  ConstantValue attribute, 103
  fields, 95, 96, 96, 96
field descriptors, 77
  CONSTANT_Fieldref_info,
  CONSTANT_Methodref_info, and
  CONSTANT_InterfaceMethodref_info
  structures, 86
  CONSTANT_NameAndType_info structure, 90
  element_value structure, 131, 132
  fields, 95, 96
  instruction representation, 160
  LocalVariableTable attribute, 125
  method type and method handle resolution, 354
  putfield, 550
  putstatic, 552
  run-time constant pool, 338
  RuntimeVisibleAnnotations attribute, 130
  structural constraints, 147
  verification type system, 155
field resolution, 351
  getfield, 442, 442
  getstatic, 444, 444
  loading constraints, 345
  method type and method handle resolution, 356
  putfield, 550, 551
  putstatic, 552, 553
  resolution, 350
fields, 95
  attributes, 100
  ClassFile structure, 73, 73
  ConstantValue attribute, 103
  Deprecated attribute, 128
RuntimeInvisibleAnnotations attribute, 133
RuntimeVisibleAnnotations attribute, 128
Signature attribute, 120
Synthetic attribute, 120
float, 232, 431
  types and the Java Virtual Machine, 26
  wide, 562
float_<n>, 233, 432
floating-point arithmetic, 19
floating-point modes, 20
d2f, 389
f2d, 419
type conversion instructions, 32
floating-point types, value sets, and values, 8
  CONSTANT_Integer_info and
  CONSTANT_Float_info structures, 87
  CONSTANT_Long_info and
  CONSTANT_Double_info structures, 89
d2f, 389, 389
f2d, 419
floating-point modes, 20
invokedynamic, 475
ldc, 516
ldc2_w, 519
ldc_w, 518
more control examples, 48
primitive types and values, 6
floating-point value set parameters, 9
  floating-point types, value sets, and values, 8, 9, 9
fmul, 234, 433
fneg, 235, 435
format checking, 140
  deriving a class from a class file
  representation, 346
format of examples, 39
format of instruction descriptions, 365
frames, 15
INDEX

aload, 373
aload_<n>, 374
anewarray, 375
areturn, 376
astore, 378
astore_<n>, 379
astore_<n>, 379
athrow, 380
checkcast, 387
dload, 401
dload_<n>, 402
dreturn, 408
dstore, 409
dstore_<n>, 410
dynamic linking, 18
fload, 431
fload_<n>, 432
freturn, 438
fstore, 439
fstore_<n>, 440
getfield, 442
getstatic, 444
iinc, 467
iload, 468
iload_<n>, 469
instanceof, 472
invokedynamic, 474
invokeinterface, 479
invokespecial, 482
invokestatic, 486
invokevirtual, 489
ireturn, 496
istore, 499
istore_<n>, 500
Java Virtual Machine stacks, 12
ldc, 515
ldc2_w, 519
ldc_w, 517

limitations of the Java Virtual Machine, 335, 335
lload, 521
lload_<n>, 522
load and store instructions, 29
local variables, 16
lreturn, 529
lstore, 532
lstore_<n>, 533
multianewarray, 541
new, 543
normal method invocation completion, 18
operand stacks, 17
pc register, 12
putfield, 550
putstatic, 552
ret, 554
return, 555
use of constants, local variables, and control constructs, 41
wide, 562
frem, 236, 436
freturn, 237, 438
fstore, 238, 439
wide, 562
fstore_<n>, 239, 440
fsub, 240, 441

g

getfield, 241, 442
getstatic, 242, 444
initialization, 359
goto, 243, 446
goto_w, 244, 447
grammar notation, 76
heap, 13

operation stacks, 17
iaload, 252, 455
iand, 253, 456
iastore, 254, 457
iconst_<i>, 458
idiv, 459
if<cond>, 257, 463
if_acmp<cond>, 255, 460
if_icmp<cond>, 256, 461
ifnonnull, 258, 465
ifnull, 259, 466
ine, 260, 467
wide, 562
iload, 261, 468
types and the Java Virtual Machine, 26
wide, 562
iload_<n>, 262, 469
imul, 263, 470
ineg, 264, 471

initialization, 359
ConstantValue attribute, 103
creation and loading, 342
getstatic, 444, 445
invokestatic, 486, 487
new, 543
preparation, 348
putstatic, 552, 553, 553
special methods, 22
InnerClasses attribute, 116
ClassFile structure, 74
instance initialization methods and newly
created objects, 331
structural constraints, 145, 146
instanceof, 265, 472
checkcast, 388
instruction representation, 159
accessors for Java Virtual Machine artifacts,
verification by type checking, 151
instruction set summary, 25
instructions, 368
static constraints, 141
integral types and values, 7
invokedynamic, 475
primitive types and values, 6
interface method resolution, 353
invokeinterface, 479, 481
loading constraints, 345
method type and method handle resolution,
internal form of names, 75
interpretation of additional tag values, 131
element_value structure, 131, 132
interpretation of fieldtype characters, 78
field descriptors, 77
verification type system, 155
introduction, 1
invokedynamic, 266, 474
BootstrapMethods attribute, 138
CONSTANT_InvokeDynamic_info
structure, 94
run-time constant pool, 339
invokeinterface, 267, 479
INDEX

invokespecial, 268, 482
   ClassFile structure, 72
   special methods, 22
invokestatic, 271, 486
   initialization, 359
invokevirtual, 272, 489
   invokedynamic, 474, 476
   invokespecial, 484
   special methods, 23
invoking methods, 50
ior, 273, 494
irem, 274, 495
ireturn, 275, 496
ishl, 276, 497
ishr, 277, 498
istore, 278, 499
   wide, 562
istore_<n>, 279, 500
isub, 280, 501
ixor, 282, 503

J
Java Virtual Machine, 2
Java Virtual Machine exit, 362
Java Virtual Machine floating-point
   arithmetic and IEEE 754, 19
Java Virtual Machine instruction set, 363
Java Virtual Machine stacks, 12
   frames, 15
Java Virtual Machine startup, 340
   initialization, 360
jsr, 504
   ret, 554, 554
   returnaddress type and values, 10
jsr_w, 505
   ret, 554
   returnaddress type and values, 10

L
l2d, 283, 506
l2f, 284, 507
l2i, 285, 508
ladd, 286, 509
laload, 287, 510
land, 288, 511
lastore, 289, 512
ldcmp, 290, 513
lconst_<l>, 291, 514
ldc, 292, 515
   call site specifier resolution, 357
   ldc_w, 518
ldc2_w, 294, 519
ldc_w, 293, 517
ldiv, 295, 520
limitations of the Java Virtual Machine, 334
   goto_w, 447
   jsr_w, 505
LineNumberTable attribute, 123
   Code attribute, 107
linking, 347
   verification of class files, 148
lload, 296, 521
   wide, 562
lload_<n>, 297, 522
lmul, 298, 523
lneg, 299, 524
load and store instructions, 29
loading constraints, 344
   creating array classes, 344
   deriving a class from a class file
   representation, 347
   field resolution, 352
   interface method resolution, 354
loading using a user-defined class loader, 343
method resolution, 353
preparation, 348
loading using a user-defined class loader, 343
creation and loading, 341
loading constraints, 345
loading using the bootstrap class loader, 342
creation and loading, 341
Java Virtual Machine startup, 340
loading constraints, 345
notation, 4
loading, linking, and initializing, 337
local variables, 16
Code attribute, 105
frames, 15
load and store instructions, 29
LocalVariableTable attribute, 124
Code attribute, 107
LocalVariableTypeTable attribute, 126
Code attribute, 107
lookupswitch, 300, 525
instruction set summary, 25
lor, 301, 527
lrem, 302, 528
lreturn, 303, 529
lshl, 304, 530
lshr, 305, 531
istore, 306, 532
wide, 562
istore_<n>, 307, 533
lsub, 308, 534
lushr, 309, 535
lxor, 310, 536

method area, 13
creation and loading, 340
run-time constant pool, 14
method descriptors, 78
areturn, 376
CONSTANT_Fieldref_info,
CONSTANT_Methodref_info, and
CONSTANT_InterfaceMethodref_info
structures, 86
CONSTANT_InvokeDynamic_info
structure, 94
CONSTANT_MethodType_info structure, 94
CONSTANT_NameAndType_info structure, 90
element_value structure, 132
invokeinterface, 479
invokestatic, 486
invokespecial, 482
invoking methods, 50
limitations of the Java Virtual Machine, 335
method type and method handle resolution, 354, 356
methods, 97, 99
RuntimeInvisibleParameterAnnotations
attribute, 137
RuntimeVisibleParameterAnnotations
attribute, 135
special methods, 22
structural constraints, 146, 146
method descriptors for method handles, 356
method type and method handle resolution, 356
method invocation and return instructions, 35
normal method invocation completion, 18
synchronization, 36
method overriding, 359
invokevirtual, 489
INDEX

preparation, 348
verification of class files, 149

method resolution, 352
invokespecial, 482, 484
invokestatic, 486, 487
invokevirtual, 489, 492
invoking methods, 52
loading constraints, 345
method type and method handle resolution, 356

method type and method handle resolution, 354
call site specifier resolution, 357, 357, 358
CONSTANT_MethodHandle_info structure, 93
initialization, 359
invokevirtual, 491, 491, 492
ldc, 515, 516
ldc_w, 517, 518
special methods, 23

methods, 97
AnnotationDefault attribute, 137
attributes, 100
ClassFile structure, 73
Code attribute, 104
constraints on Java Virtual Machine code, 140
Deprecated attribute, 128
Exceptions attribute, 115
floating-point modes, 20
getfield, 442
invokespecial, 482
invokevirtual, 489
method type and method handle resolution, 356
putfield, 550
RuntimeInvisibleParameterAnnotations attribute, 133

RuntimeInvisibleParameterAnnotations attribute, 136
RuntimeVisibleAnnotations attribute, 128
RuntimeVisibleParameterAnnotations attribute, 134
Signature attribute, 120
special methods, 22
synchronization, 36
Synthetic attribute, 120

mnemonic, 366
monitorenter, 311, 537
invokeinterface, 480
invokespecial, 483
invokestatic, 486
invokevirtual, 490
monitorexit, 539

monitorexit, 312, 539
areturn, 376
athrow, 380
dreturn, 408
freturn, 438
invokeinterface, 480
invokespecial, 484
invokestatic, 487
invokevirtual, 490
ireturn, 496
lreturn, 529
monitorenter, 537
return, 555

more control examples, 47
dcmp<op>, 396
fcmp<op>, 426

multianewarray, 313, 541

N

native method stacks, 14
native methods
invokeinterface, 480
invokespecial, 483
invokestatic, 487
invokevirtual, 490

nested class access and property flags, 118
  InnerClasses attribute, 118, 118

new, 314, 543
  initialization, 359
  StackMapTable attribute, 115
newarray, 315, 545
  baload, 382
  bastore, 383
  boolean type, 10
  multianewarray, 542

nop, 316, 547

normal method invocation completion, 18
  synchronization, 67

notation, 4

O

Object creation and manipulation, 34
  load and store instructions, 29
opcode mnemonics by opcode, 565
operand stack, 415, 416, 417, 414, 549
operand stack management instructions, 34
operand stacks, 17
  Code attribute, 105
  format of instruction descriptions, 367
  frames, 15
  load and store instructions, 29
  structural constraints, 145, 145
operations on the operand stack, 58
overriding
  invokevirtual, 489
  preparation, 348
  verification of class files, 149

P

pc register, 12
pop, 317, 548
pop2, 318, 549
predefined class file attributes, 102
  attributes, 101
preparation, 348
  loading constraints, 345, 345
primitive types and values, 6
  multianewarray, 541
  new, 543
  newarray, 545
  preparation, 348
process of verification by type inference, 327
public design, private implementation, 37
  reserved opcodes, 364
putfield, 319, 550
putstatic, 320, 552
  initialization, 359

R

receiving arguments, 49
  invoking methods, 52
reference types and values, 11
  anewarray, 375
  control transfer instructions, 35
  field resolution, 352
  interface method resolution, 354, 354
  method resolution, 353, 353
  multianewarray, 541
  new, 543
  newarray, 545
  preparation, 348, 348, 348, 349, 349
representation of objects, 19
reserved opcodes, 364
  static constraints, 141

581
INDEX

resolution, 349
  creation and loading, 342
  loading constraints, 345
ret, 554
  jsr, 504
  jsr_w, 505
  return address type and values, 10
  wide, 562
return, 321, 555
  ret, 554
return address type and values, 10
  primitive types and values, 6
run-time constant pool, 14, 337
  dynamic linking, 18
  frames, 15
  getfield, 442
  getstatic, 444
  invokedynamic, 474, 474, 476
  invokeinterface, 479
  invokespecial, 482
  invokestatic, 486
  invokevirtual, 489
  ldc, 515, 515, 515, 515
  ldc2_w, 519
  ldc_w, 517, 517, 517
  method type and method handle resolution, 355
  putfield, 550
  putstatic, 552
  resolution, 350
  run-time constant pool, 337
run-time data areas, 11
RuntimeInvisibleAnnotations attribute, 133
  annotations, 67, 67
  ClassFile structure, 74, 74
  fields, 96, 96
  methods, 100, 100
RuntimeInvisibleParameterAnnotations attribute, 136
  methods, 100, 100
RuntimeVisibleAnnotations attribute, 128
  annotations, 67, 67
  ClassFile structure, 74, 74
  fields, 96, 96
  methods, 100, 100
RuntimeVisibleParameterAnnotations attribute, 134
  methods, 100, 100
S
saload, 322, 556
sastore, 323, 557
Signature attribute, 120
  ClassFile structure, 74, 74
  fields, 96, 96
  methods, 100, 100
signatures, 79
  LocalVariableTypeTable attribute, 127
  Signature attribute, 121
sipush, 324, 558
SourceDebugExtension attribute, 122
  ClassFile structure, 74
SourceFile attribute, 121
  ClassFile structure, 74
special methods, 22
  ClassFile structure, 74
  Code attribute, 104
  CONSTANT_Fieldref_info,
  CONSTANT_Methodref_info, and
  CONSTANT_InterfaceMethodref_info
  structures, 86
  CONSTANT_MethodHandle_info structure, 93
CONSTANT_NameAndType_info structure, 90
ConstantValue attribute, 103
constraints on Java Virtual Machine code, 140
initialization, 359
instance initialization methods and newly created objects, 331
invokeinterface, 479
invokespecial, 482, 484
invokestatic, 486
invokevirtual, 489, 491
method area, 13
method invocation and return instructions, 35
method resolution, 352
methods, 97, 99, 99, 99
new, 544
putfield, 550
putstatic, 552
static constraints, 143
structural constraints, 145, 145, 146
Synthetic attribute, 120
unqualified names, 75
working with class instances, 53
stack map frame representation, 160
accessors for Java Virtual Machine artifacts, 153
type checking load and store instructions, 174, 174
verification by type checking, 151
StackMapTable attribute, 107
Code attribute, 107, 107
stack map frame representation, 161
verification by type checking, 150
startup
initialization, 360
static constraints, 141
limitations of the Java Virtual Machine, 336
structural constraints, 144
limitations of the Java Virtual Machine, 336
structure of the Java Virtual Machine, 5
summary of chapters, 3
swap, 325, 559
operand stacks, 17
synchronization, 36, 66
areturn, 376, 376
athrow, 380, 381
dreturn, 408, 408
freturn, 438, 438
ireturn, 496, 496
lreturn, 529, 529
monitorenter, 537
monitorexit, 539, 540, 539
return, 555, 555
synchronization, 36, 66
Synthetic attribute, 120
ClassFile structure, 74
fields, 96
methods, 98, 100
T
tableswitch, 326, 560
instruction set summary, 25
throwing and handling Exceptions, 59
Exceptions, 25
more control examples, 47
throwing Exceptions, 36
type checking abstract and native methods, 166
verification by type checking, 151
type checking for protected members, 176
getfield, 241
invokevirtual, 272
verification by type checking, 152
type checking instructions, 179
INDEX

stack map frame representation, 164
verification by type checking, 152

**type checking load and store instructions**, 174
stack map frame representation, 162
verification by type checking, 152

**type checking methods with Code**, 167
StackMapTable attribute, 109
verification by type checking, 151

**type conversion instructions**, 32

**type support in the Java Virtual Machine instruction set**, 28

- types and the Java Virtual Machine, 26, 26
- use of constants, local variables, and control constructs, 43

**types and the Java Virtual Machine**, 26

- arithmetic instructions, 30
- control transfer instructions, 35
- data types, 6
dup, 412
dup_x1, 413
load and store instructions, 30
operand stack, 415, 415, 416, 416, 417, 417, 417, 417, 414, 414, 549, 549
pop, 548
stack map frame representation, 164
structural constraints, 145
swap, 559
type conversion instructions, 32

**V**

**value set conversion**, 20
d2f, 389
d2i, 390
d2l, 391
dadd, 392
dastore, 395
dcmp<op>, 396
ddiv, 399
dneg, 405
drem, 406
dreturn, 408
dstore, 409
dstore_<n>, 410
dsub, 411
f2d, 419
f2i, 420
f2l, 421
fadd, 422
fastore, 425
fcmp<op>, 426
fdiv, 429
floating-point modes, 20
fmul, 433
fneg, 435

**U**

**unqualified names**, 75

- binary class and interface names, 75
CONSTANT_NameAndType_info structure, 90
fields, 96
LocalVariableTable attribute, 125
values of types long and double, 330
verification, 348
  access control, 358
  creation and loading, 342
verification by type checking, 149
  StackMapTable attribute, 107
verification by type inference, 327
  verification by type checking, 149
verification of class files, 148
  assumptions: the meaning of "must", 363
  operand stacks, 17
  verification, 348
verification type system, 155
  accessors for Java Virtual Machine artifacts, 153
  stack map frame representation, 162
  StackMapTable attribute, 108
  verification by type checking, 151
Virtual Machine errors, 364
  Exceptions, 24

W
wide, 327, 562
 aload, 373
  astore, 378
  dload, 401

dstore, 409
fload, 431
fstore, 439
iinc, 467
iload, 468
istore, 499
lload, 521
lstore, 532
ret, 554

working with class instances, 53
  accessing the run-time constant pool, 46
  invoking methods, 52