CHAPTER 3

The Structure of the Java Virtual Machine

THIS book specifies an abstract machine. It does not document any particular implementation of the Java virtual machine, including Sun Microsystems'.

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

3.1 The `class` File Format

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

Chapter 4, "The `class` File Format," covers the `class` file format in detail.

3.2 Data Types

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

The Java virtual machine expects that nearly all type checking is done prior to 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 §3.11.1.

The Java virtual machine contains explicit support for objects. An object is either a dynamically allocated class instance or an array. A reference to an object is considered to have Java virtual machine type `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`.

3.3 Primitive Types and Values

The primitive data types supported by the Java virtual machine are the *numeric types*, the `boolean` type (§3.3.4),[1] and the `returnAddress` type (§3.3.3). The numeric types consist of the *integral types* (§3.3.1) and the *floating-point types* (§3.3.2). The integral types are:

- `byte`, whose values are 8-bit signed two's-complement integers
- `short`, whose values are 16-bit signed two's-complement integers
- `int`, whose values are 32-bit signed two's-complement integers
- `long`, whose values are 64-bit signed two's-complement integers
- `char`, whose values are 16-bit unsigned integers representing UTF-16 code units (§2.1)

[1] The first edition of *The Java™ Virtual Machine Specification* did not consider `boolean` to be a Java virtual machine type. However, `boolean` values do have limited support in the Java virtual machine. This second edition clarifies the issue by treating `boolean` as a type.

The floating-point types are:

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

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

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

3.3.1 Integral Types and Values

The values of the integral types of the Java virtual machine are the same as those for the integral types of the Java programming language (§2.4.1):

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

3.3.2 Floating-Point Types, Value Sets, and Values

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

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

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

The finite nonzero values of any floating-point value set can all be expressed in the form $s \cdot m \cdot 2^{(e - N + 1)}$, where s is +1 or −1, m is a positive integer less than 2^N, and e is an integer between $Emin = -(2^{K-1} - 2)$ and $Emax = 2^{K-1} - 1$, 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 $Emin$ and $Emax$) for the two required and two optional floating-point value sets are summarized in Table 3.1.

Table 3.1 Floating-point value set parameters

Parameter	**float**	**float-extended-exponent**	**double**	**double-extended-exponent**
N	24	24	53	53
K	8	≥ 11	11	≥ 15
Emax	+127	≥ +1023	+1023	≥ +16383
Emin	−126	≤ −1022	−1022	≤ −16382

Where one or both extended-exponent value sets are supported by an implementation, then for each supported extended-exponent value set there is a specific implementation-dependent constant K, whose value is constrained by Table 3.1; this value K in turn dictates the values for $Emin$ and $Emax$.

Each of the four value sets includes not only the finite nonzero values that are ascribed to it above, but also the five values positive zero, negative zero, positive infinity, negative infinity, and NaN.

Note that the constraints in Table 3.1 are designed so that every element of the float value set is necessarily also an element of the float-extended-exponent value set, the double value set, and the double-extended-exponent value set. Likewise, each element of the double value set is necessarily also an element of the double-extended-exponent value set. Each extended-exponent value set has a larger range of exponent values than the corresponding standard value set, but does not have more precision.

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

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

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

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

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

3.3.3 The `returnAddress` Type and Values

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

3.3.4 The `boolean` Type

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

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

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

3.4 Reference Types and Values

There are three kinds of `reference` types: class types, array types, and interface types. Their values are references to dynamically created class instances, arrays, or class instances or arrays that implement interfaces, respectively. A `reference` value may also be the special null reference, a reference to no object, which will be denoted here by `null`. The `null` reference initially has no runtime type, but may be cast to any type (§2.4).

[2] In Sun's JDK releases 1.0 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, `boolean` arrays in the Java programming language are encoded as Java virtual machine `byte` arrays, using 8 bits per `boolean` element.

The Java virtual machine specification does not mandate a concrete value encoding `null`.

3.5 Runtime Data Areas

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

3.5.1 The `pc` Register

The Java virtual machine can support many threads of execution at once (§2.19). 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, the current method (§3.6) for that thread. If that method is not `native`, the `pc` register contains the address of the Java virtual machine instruction currently being executed. If the method currently being executed by the thread is `native`, the value of the Java virtual machine's `pc` register is undefined. The Java virtual machine's `pc` register is wide enough to hold a `returnAddress` or a native pointer on the specific platform.

3.5.2 Java Virtual Machine Stacks

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

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

[3] In the first edition of this specification, the Java virtual machine stack was known as the *Java stack*.

created. A Java virtual machine implementation may provide the programmer or the user control over the initial size of Java virtual machine stacks, as well as, in the case of dynamically expanding or contracting Java virtual machine stacks, control over the maximum and minimum sizes.[4]

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

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

3.5.3 Heap

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

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

A Java virtual machine implementation may provide the programmer or the user control over the initial size of the heap, as well as, if the heap can be dynami-

[4] In Sun's implementations of the Java virtual machine in JDK releases 1.0.2 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, Java virtual machine stacks are discontiguous and are independently expanded as required by the computation. Those implementations do not free memory allocated for a Java virtual machine stack until the associated thread terminates. Expansion is subject to a size limit for any one stack. The Java virtual machine stack size limit may be set on virtual machine start-up using the "`-oss`" flag. The Java virtual machine stack size limit can be used to limit memory consumption or to catch runaway recursions.

cally expanded or contracted, control over the maximum and minimum heap size.[5]

The following exceptional condition is associated with the heap:

- If a computation requires more heap than can be made available by the automatic storage management system, the Java virtual machine throws an `OutOfMemoryError`.

3.5.4 Method Area

The Java virtual machine has a *method area* that is shared among all Java virtual machine threads. The method area is analogous to the storage area for compiled code of a conventional language or analogous to the "text" segment in a UNIX process. It stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods (§3.9) used in class and instance initialization and interface type 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.[6]

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.`

---

[5] Sun's implementations of the Java virtual machine in JDK releases 1.0.2 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, dynamically expand the heap as required by the computation, but never contract the heap. The initial and maximum sizes may be specified on virtual machine start-up using the "`-ms`" and "`-mx`" flags, respectively.

[6] Sun's implementation of the Java virtual machine in JDK release 1.0.2 dynamically expands the method area as required by the computation, but never contracts the method area. The Java virtual machine implementations in Sun's JDK release 1.1 and the Java 2 SDK, Standard Edition, v1.2 garbage collect the method area. In neither case is user control over the initial, minimum, or maximum size of the method area provided.

3.5.5 Runtime Constant Pool

A *runtime constant pool* is a per-class or per-interface runtime representation of the `constant_pool` table in a `class` file (§4.4). It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at run time. The runtime constant pool serves a function similar to that of a symbol table for a conventional programming language, although it contains a wider range of data than a typical symbol table.

Each runtime constant pool is allocated from the Java virtual machine's method area (§3.5.4). The runtime constant pool for a class or interface is constructed when the class or interface is created (§5.3) by the Java virtual machine.

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

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

See Chapter 5 for information about the construction of the runtime constant pool.

3.5.6 Native Method Stacks

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

The Java virtual machine specification permits native method stacks either to be of a fixed size or to dynamically expand and contract as required by the computation. If the native method stacks are of a fixed size, the size of each native method stack may be chosen independently when that stack is created. In any case, a Java virtual machine implementation may provide the programmer or the user control over the initial size of the native method stacks. In the case of varying-size native method stacks, it may also make available control over the maximum and minimum method stack sizes.[7]

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 `OutOfMemory-Error`.

3.6 Frames

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

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

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

Only one frame, the frame for the executing method, is active at any point in a given thread of control. This frame is referred to as the *current frame*, and its method is known as the *current method*. The class in which the current method is

[7] Sun's implementations of the Java virtual machine in JDK releases 1.0.2 and 1.1, and the Java 2 SDK, Standard Edition, v1.2, allocate fixed-size native method stacks of a single size. The size of the native method stacks may be set on virtual machine start-up using the "`-ss`" flag. The native method stack size limit can be used to limit memory consumption or to catch runaway recursions in `native` methods. Sun's implementations do *not* check for native method stack overflow.

defined is the *current class*. Operations on local variables and the operand stack are typically with reference to the current frame.

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

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

3.6.1 Local Variables

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

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

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

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

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

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

3.6.2 Operand Stacks

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

Where it is clear by context, we will sometimes refer to the operand stack of the current frame as simply the operand stack.

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

For example, the *iadd* instruction 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 and *swap*) operate on runtime data areas as raw values without regard to their specific types; these instructions are defined in such a way that they cannot be used to modify or break up individual values. These restrictions on operand stack manipulation are enforced through `class` file verification (§4.9).

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.

3.6.3 Dynamic Linking

Each frame (§3.6) contains a reference to the runtime constant pool (§3.5.5) for the type of the current method to support *dynamic linking* of the method code. The `class` file code for a method refers to methods to be invoked and variables to be accessed via symbolic references. Dynamic linking translates these symbolic method references into concrete method references, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.

This late binding of the methods and variables makes changes in other classes that a method uses less likely to break this code.

3.6.4 Normal Method Invocation Completion

A method invocation *completes normally* if that invocation does not cause an exception (§2.16, §3.10) 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 (§3.11.8), the choice of which must be appropriate for the type of the value being returned (if any).

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

3.6.5 Abrupt Method Invocation Completion

A method invocation *completes abruptly* if execution of a Java virtual machine instruction within the method causes the Java virtual machine to throw an exception (§2.16, §3.10), and that exception is not handled within the method. Execution of an *athrow* instruction also causes an exception to be explicitly thrown and, if the exception is not caught by the current method, results in abrupt method invocation completion. A method invocation that completes abruptly never returns a value to its invoker.

3.6.6 Additional Information

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

3.7 Representation of Objects

The Java virtual machine does not mandate any particular internal structure for objects.[8]

3.8 Floating-Point Arithmetic

The Java virtual machine incorporates a subset of the floating-point arithmetic specified in *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985, New York).

3.8.1 Java Virtual Machine Floating-Point Arithmetic and IEEE 754

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

- The floating-point operations of the Java virtual machine do not throw exceptions, trap, or otherwise signal the IEEE 754 exceptional conditions of invalid operation, division by zero, overflow, underflow, or inexact. The Java virtual machine has no signaling NaN value.
- The Java virtual machine does not support IEEE 754 signaling floating-point comparisons.
- The rounding operations of the Java virtual machine always use IEEE 754 round to nearest mode. Inexact results are rounded to the nearest representable value, with ties going to the value with a zero least-significant bit. This is the IEEE 754 default mode. But Java virtual machine instructions that convert

[8] In some of Sun's implementations of the Java virtual machine, a reference to a class instance is a pointer to a *handle* that is itself a pair of pointers: one to a table containing the methods of the object and a pointer to the `Class` object that represents the type of the object, and the other to the memory allocated from the heap for the object data.

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.

3.8.2 Floating-Point Modes

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

Note that this mapping of the `ACC_STRICT` bit implies that methods in classes compiled by a compiler that predates the Java 2 platform, v1.2, are effectively not FP-strict.

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

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

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

3.8.3 Value Set Conversion

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

Where value set conversion is indicated, an implementation is permitted to perform one of the following operations on a value:

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

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

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

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

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

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

3.9 Specially Named Initialization Methods

At the level of the Java virtual machine, every constructor (§2.12) appears as an *instance initialization method* that has the special name `<init>`. This name is supplied by a compiler. Because the name `<init>` is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Instance initialization methods may be invoked only within the Java virtual machine by the *invokespecial* instruction, and they may be invoked only on uninitialized class instances. An instance initialization method takes on the access permissions (§2.7.4) of the constructor from which it was derived.

A class or interface has at most one *class or interface initialization method* and is initialized (§2.17.4) by invoking that method. The initialization method of a class or interface is static and takes no arguments. It has the special name `<clinit>`. This name is supplied by a compiler. Because the name `<clinit>` is not a valid identifier, it cannot be used directly in a program written in the Java programming language. Class and interface initialization methods are invoked implicitly by the Java virtual machine; they are never invoked directly from any Java virtual machine instruction, but are invoked only indirectly as part of the class initialization process.

3.10 Exceptions

In the Java programming language, throwing an exception results in an immediate nonlocal transfer of control from the point where the exception was thrown. This transfer of control may abruptly complete, one by one, multiple statements, constructor invocations, static and field initializer evaluations, and method invocations. The process continues until a `catch` clause (§2.16.2) is found that handles the thrown value. If no such clause can be found, the current thread exits.

In cases where a `finally` clause (§2.16.2) is used, the `finally` clause is executed during the propagation of an exception thrown from the associated `try` block and any associated `catch` block, even if no `catch` clause that handles the thrown exception may be found.

As implemented by the Java virtual machine, each `catch` or `finally` clause of a method is represented by an exception handler. 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 (§3.6.5). On abrupt completion, the operand stack and local variables of the current method invocation are discarded, and its frame is popped, reinstating the frame of the invoking method. The exception is then rethrown in the context of the invoker's frame and so on, continuing up the method invocation chain. If no suitable exception handler is found before the top of the method invocation chain is reached, the execution of the thread in which the exception was thrown is terminated.

The order in which the exception handlers of a method are searched for a match is important. Within a `class` file the exception handlers for each method are stored in a table (§4.7.3). At 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. Because `try` statements are structured, a compiler for the Java programming language can always order the entries of the exception handler table such that, for any thrown exception and any program counter value in that method, the first exception handler that matches the thrown exception corresponds to the innermost matching `catch` or `finally` clause.

Note that the Java virtual machine does not enforce nesting of or any ordering of the exception table entries of a method (§4.9.5). The exception handling semantics of the Java programming language are implemented only through cooperation with the compiler. When `class` files are generated by some other means, the defined search procedure ensures that all Java virtual machines will behave consistently.

More information on the implementation of `catch` and `finally` clauses is given in Chapter 7, "Compiling for the Java Virtual Machine."

3.11 Instruction Set Summary

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

Ignoring exceptions, the inner loop of a Java virtual machine interpreter is effectively

```
do {
    fetch an opcode;
    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

$$(byte1 << 8) \mid byte2$$

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

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

3.11.1 Types and the Java Virtual Machine

Most of the instructions in the Java virtual machine instruction set encode type information about the operations they perform. For instance, the *iload* instruction loads the contents of a local variable, which must be an `int`, onto the operand stack. The *fload* instruction does the same with a `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: *i* for an `int` operation, *l* for `long`, *s* for `short`, *b* for `byte`, *c* for `char`, *f* for `float`, *d* for `double`, and *a* for `reference`. Some instructions for which the type is unambiguous do not have a type letter in their mnemonic. For instance, *arraylength* always operates on an object that is an array. 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 runtime data types, there would be more instructions than could be represented in a byte. Instead, the instruction set of the Java virtual machine provides a reduced level of type support for certain operations. In other words, the instruction set is intentionally not orthogonal. Separate instructions can be used to convert between unsupported and supported data types as necessary.

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

Note that most instructions in Table 3.2 do not have forms for the integral types `byte`, `char`, and `short`. None have forms for the `boolean` type. Compilers encode loads of literal values of types `byte` and `short` using Java virtual machine instructions that sign-extend those values to values of type `int` at compile time or 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 3.2 Type support in the Java virtual machine instruction set

opcode	`byte`	`short`	`int`	`long`	`float`	`double`	`char`	`reference`
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			*iushr*	*lushr*				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

The mapping between Java virtual machine actual types and Java virtual machine computational types is summarized by Table 3.3.

Table 3.3 Java virtual machine actual and computational types

Actual Type	**Computational Type**	**Category**
`boolean`	`int`	category 1
`byte`	`int`	category 1
`char`	`int`	category 1
`short`	`int`	category 1
`int`	`int`	category 1
`float`	`float`	category 1
`reference`	`reference`	category 1
`returnAddress`	`returnAddress`	category 1
`long`	`long`	category 2
`double`	`double`	category 2

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

The remainder of this chapter summarizes the Java virtual machine instruction set.

3.11.2 Load and Store Instructions

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

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

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

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

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

3.11.3 Arithmetic Instructions

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

- Add: *iadd*, *ladd*, *fadd*, *dadd*.
- Subtract: *isub*, *lsub*, *fsub*, *dsub*.
- Multiply: *imul*, *lmul*, *fmul*, *dmul*.
- Divide: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Remainder: *irem*, *lrem*, *frem*, *drem*.
- Negate: *ineg*, *lneg*, *fneg*, *dneg*.
- Shift: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.

- Bitwise OR: *ior*, *lor*.
- Bitwise AND: *iand*, *land*.
- Bitwise exclusive OR: *ixor*, *lxor*.
- Local variable increment: *iinc*.
- Comparison: *dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

The semantics of the Java programming language operators on integer and floating-point values (§2.4.2, §2.4.4) are directly supported by the semantics of the Java virtual machine instruction set.

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

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

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

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

The Java virtual machine's floating-point operators do not throw runtime exceptions (not to be confused with IEEE 754 floating-point exceptions). An operation that overflows produces a signed infinity, an operation that underflows produces a denormalized value or a signed zero, and an operation that has no mathematically definite result produces NaN. All numeric operations with NaN as an operand produce NaN as a result.

Comparisons on values of type `long` (*lcmp*) perform a signed comparison. Comparisons on values of floating-point types (*dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*) are performed using IEEE 754 nonsignaling comparisons.

3.11.4 Type Conversion Instructions

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

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

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

The 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 (§3.8.2) also preserve the numeric value exactly; however, such conversions that are not FP-strict may lose information about the overall magnitude of the converted value.

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

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

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

Note that widening numeric conversions do not exist from integral types `byte`, `char`, and `short` to type `int`. As noted in §3.11.1, values of type `byte`, `char`, and `short` are internally widened to type `int`, making these conversions implicit.

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

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

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

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

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

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

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

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

3.11.5 Object Creation and Manipulation

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

- Create a new class instance: *new*.
- Create a new array: *newarray*, *anewarray*, *multianewarray*.
- Access fields of classes (`static` fields, known as class variables) and fields of class instances (non-`static` fields, known as instance variables): *getfield*, *putfield*, *getstatic*, *putstatic*.
- Load an array component onto the operand stack: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Store a value from the operand stack as an array component: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- Get the length of array: *arraylength*.
- Check properties of class instances or arrays: *instanceof*, *checkcast*.

3.11.6 Operand Stack Management Instructions

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

3.11.7 Control Transfer Instructions

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

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

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

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

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

3.11.8 Method Invocation and Return Instructions

The following four instructions invoke methods:

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

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.

3.11.9 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.

3.11.10 Implementing `finally`

The implementation of the `finally` keyword uses the *jsr*, *jsr_w*, and *ret* instructions. See Section 4.9.6, "Exceptions and `finally`," and Section 7.13, "Compiling `finally`."

3.11.11 Synchronization

The Java virtual machine supports synchronization of both methods and sequences of instructions within a method using a single synchronization construct: the *monitor.*

Method-level synchronization is handled as part of method invocation and return (see Section 3.11.8, "Method Invocation and Return Instructions").

Synchronization of sequences of instructions is typically used to encode the synchronized blocks of the Java programming language. The Java virtual machine supplies the *monitorenter* and *monitorexit* instructions to support such constructs.

Proper implementation of synchronized blocks requires cooperation from a compiler targeting the Java virtual machine. The compiler must ensure that at any method invocation completion a *monitorexit* instruction will have been executed for each *monitorenter* instruction executed since the method invocation. This must be the case whether the method invocation completes normally (§3.6.4) or abruptly (§3.6.5).

The compiler enforces proper pairing of *monitorenter* and *monitorexit* instructions on abrupt method invocation completion by generating exception handlers (§3.10) that will match any exception and whose associated code executes the necessary *monitorexit* instructions (§7.14).

3.12 Class Libraries

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

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

- Reflection, such as the classes in the package `java.lang.reflect` and the class `Class`.
- Loading and creation of a class or interface. The most obvious example is the class `ClassLoader`.
- Linking and initialization of a class or interface. The example classes cited above fall into this category as well.
- Security, such as the classes in the package `java.security` and other classes such as `SecurityManager`.
- Multithreading, such as the class `Thread`.
- Weak references, such as the classes in the package `java.lang.ref`.[9]

The list above is meant to be illustrative rather than comprehensive. An exhaustive list of these classes or of the functionality they provide is beyond the scope of this book. See the specifications of the Java and Java 2 platform class libraries for details.

3.13 Public Design, Private Implementation

Thus far this book has sketched the public view of the Java virtual machine: the `class` file format and the instruction set. These components are vital to the hardware-, operating system-, and implementation-independence of the Java virtual machine. The implementor may prefer to think of them as a means to securely communicate fragments of programs between hosts each implementing the Java or Java 2 platform, rather than as a blueprint to be followed exactly.

[9] Weak references were introduced in the Java 2 platform, v1.2.

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.[10]

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.

[10] There are some exceptions: debuggers, profilers, and just-in-time code generators can each require access to elements of the Java virtual machine that are normally considered to be "under the hood." Where appropriate, Sun is working with other Java virtual machine implementors and tools vendors to develop common interfaces to the Java virtual machine for use by such tools, and to promote those interfaces across the industry. Information on publicly available low-level interfaces to the Java virtual machine will be made available at `http://java.sun.com`.