



# 本科毕业设计（论文）

**外文参考文献译文及原文**

**学 院 计算机学院**

**专 业 网络工程**

**年级班别 2015级（4）班**

**学 号 3115005534**

**学生姓名 李楚豪**

**指导教师 张海笑**

**2019年 5 月**

目录

[1. 中文译文 1](#_Toc16900)

[1.1 class文件格式 1](#_Toc15402)

[1.2 数据类型 1](#_Toc17899)

[1.3 原始类型和价值观 2](#_Toc8768)

[1.3.1 积分类型和值 2](#_Toc14826)

[1.3.2 浮点类型，值集和值 3](#_Toc24735)

[1.3.3 returnAddress类型和值 5](#_Toc2131)

[1.3.4 boolean类型 5](#_Toc20207)

[1.4 参考类型和值 5](#_Toc20065)

[1.5 运行时数据区 6](#_Toc28903)

[1.5.1 pc注册 6](#_Toc5811)

[1.5.2 Java虚拟机堆栈 6](#_Toc28006)

[1.5.3 堆 7](#_Toc126)

[1.5.4 方法区 8](#_Toc29523)

[1.5.5 运行时常量池 8](#_Toc360)

[1.5.6 本机方法堆栈 8](#_Toc8855)

[1.6 框架 9](#_Toc7984)

[1.6.1局部变量 10](#_Toc418)

[1.6.2操作数堆栈 11](#_Toc10119)

[1.6.3动态链接 11](#_Toc3162)

[1.6.4正常方法调用完成 12](#_Toc24521)

[1.6.5突然的方法调用完成 12](#_Toc3956)

[1.7 对象的表示 12](#_Toc13180)

[1.8浮点运算 12](#_Toc23416)

[1.8.1 Java虚拟机浮点运算和IEEE 754 13](#_Toc24506)

[1.8.2 浮点模式 13](#_Toc29954)

[1.8.3价值集转换 14](#_Toc19185)

[1.9特殊方法 15](#_Toc21506)

[1.10例外 16](#_Toc29598)

[2. 英文原文 18](#_Toc4619)

[2.1 The class File Format 18](#_Toc30194)

[2.2 Data Types 18](#_Toc10216)

[2.3 Primitive Types and Values 19](#_Toc26491)

[2.3.1 Integral Types and Values 20](#_Toc7442)

[2.3.2 Floating-Point Types, Value Sets, and Values 20](#_Toc334)

[2.3.3 The returnAddress Type and Values 23](#_Toc24518)

[2.3.4 The boolean Type 24](#_Toc5188)

[2.4 Reference Types and Values 24](#_Toc29382)

[2.5 Run-Time Data Areas 25](#_Toc24640)

[2.5.1 The pc Register 25](#_Toc17921)

[2.5.2 Java Virtual Machine Stacks 25](#_Toc28322)

[2.5.3 Heap 26](#_Toc31170)

[2.5.4 Method Area 27](#_Toc20814)

[2.5.5 Run-Time Constant Pool 28](#_Toc2067)

[2.5.6. Native Method Stacks 28](#_Toc29427)

[2.6. Frames 29](#_Toc28278)

[2.6.1. Local Variables 30](#_Toc19058)

[2.6.2. Operand Stacks 31](#_Toc547)

[2.6.3. Dynamic Linking 32](#_Toc4756)

[2.6.4. Normal Method Invocation Completion 33](#_Toc11109)

[2.6.5. Abrupt Method Invocation Completion 33](#_Toc28792)

[2.7. Representation of Objects 34](#_Toc26841)

[2.8. Floating-Point Arithmetic 34](#_Toc1627)

[2.8.1. Java Virtual Machine Floating-Point Arithmetic and IEEE 754 34](#_Toc16786)

[2.8.2. Floating-Point Modes 35](#_Toc7591)

[2.8.3. Value Set Conversion 36](#_Toc17776)

[2.9. Special Methods 37](#_Toc10856)

[2.10. Exceptions 39](#_Toc30004)

# 中文译文

本文档指定了一个抽象机器。它没有描述Java虚拟机的任何特定实现。

要正确实现Java虚拟机，您只需要能够读取class文件格式并正确执行其中指定的操作。不属于Java虚拟机规范的实现细节会不必要地限制实现者的创造力。例如，运行时数据区域的内存布局，使用的垃圾收集算法以及Java虚拟机指令的任何内部优化（例如，将它们转换为机器代码）由实现者自行决定。

本规范中对Unicode的所有引用都是针对Unicode标准版本6.0.0提供的，可从以下位置获得 http://www.unicode.org/。

## 1.1 class文件格式

由Java虚拟机执行的编译代码使用独立于硬件和操作系统的二进制格式表示，通常（但不一定）存储在文件中，称为class文件格式。该class文件格式精确定义的类或接口，其中包括详细信息，如字节顺序理所当然在特定平台的目标文件格式可能采取的代表性。

第4章“ class文件格式” class详细介绍了文件格式。

## 1.2 数据类型

与Java编程语言一样，Java虚拟机也可以使用两种类型：基本类型 和引用类型。相应地，有两种值可以存储在变量中，作为参数传递，由方法返回，并对其进行操作：原始值和参考值。

Java虚拟机期望几乎所有类型检查都在运行时之前完成，通常由编译器完成，而不必由Java虚拟机本身完成。原始类型的值不需要被标记或以其他方式可检查以在运行时确定它们的类型，或者与引用类型的值区分。相反，Java虚拟机的指令集使用旨在对特定类型的值进行操作的指令来区分其操作数类型。例如，iadd，ladd，fadd和 dadd都是Java虚拟机指令添加两个数值，并产生数值结果，但每个专业的操作数类型： int，long，float，和double分别。有关Java虚拟机指令集中类型支持的摘要，请参见 [§2.11.1](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.11.1" \o "2.11.1。 类型和Java虚拟机)。

Java虚拟机包含对对象的显式支持。对象是动态分配的类实例或数组。对对象的引用被视为具有Java虚拟机类型reference。类型的值reference可以被认为是指向对象的指针。可能存在多个对象的引用。始终通过类型值操作，传递和测试对象 reference。

## 1.3 原始类型和价值观

Java虚拟机支持的原始数据类型是数字类型， boolean类型（[§2.3.4](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.4" \o "2.3.4。 布尔类型)）和returnAddress 类型（[§2.3.3](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.3" \o "2.3.3。 returnAddress类型和值)）。

数字类型由整数类型（[§2.3.1](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.1" \o "2.3.1。 积分类型和值)）和浮点类型 （[§2.3.2](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2" \o "2.3.2。 浮点类型，值集和值)）组成。

整体类型是：

byte，其值为8位有符号二进制补码整数，其默认值为零

short，其值为16位有符号二进制补码整数，其默认值为零

int，其值为32位有符号二进制补码整数，其默认值为零

long，其值为64位有符号二进制补码整数，其默认值为零

char，其值为16位无符号整数，表示基本多语言平面中的Unicode代码点，使用UTF-16编码，其默认值为空代码点（'\u0000'）

浮点类型是：

float，其值是浮点值集的元素，或者，如果支持，则为float-extended-exponent值集，其默认值为正零

double，其值是double值集的元素，或者，如果支持，则为double-extended-exponent值集，其默认值为正零

所述的值boolean 类型编码的真值true和false，并且缺省值是false。

第一版的的Java ®虚拟机规范并不认为 boolean是一个Java虚拟机类型。但是，boolean值在Java虚拟机中的支持有限。第二版的的Java ®虚拟机规范 的治疗澄清这个问题boolean作为一个类型。

该returnAddress类型的值 是指向Java虚拟机指令的操作码的指针。在原始类型中，只有returnAddress类型不与Java编程语言类型直接相关。

### 1.3.1 积分类型和值

Java虚拟机的整数类型的值是：

对于byte，从-128到127（-2 7到2 7 - 1），包括在内

对于short，从-32768到32767（-2 15到2 15 - 1），包括在内

对于int，从-2147483648到2147483647（-2 31到2 31 - 1），包括在内

对于long，从-9223372036854775808到9223372036854775807（-2 63到2 63 - 1），包括

适用char于0到65535（含）

### 1.3.2 浮点类型，值集和值

浮点类型是 float和double，它在概念上与32位单精度和64位双精度格式IEEE 754值和操作相关联，如IEEE标准二进制浮点运算（ANSI / IEEE标准。 754-1985，纽约）。

IEEE 754标准不仅包括正负符号幅度数，还包括正负零，正无穷大和负无穷大，以及特殊的非数字值（以下简称为“NaN”）。NaN值用于表示某些无效操作的结果，例如将零除以零。

Java虚拟机的每个实现都需要支持两组标准的浮点值，称为浮点值集和双值集。此外，Java虚拟机的实现可以选择支持两个扩展指数浮点值集中的一个或两个，称为 float-extended-exponent值集和double-extended-exponent值集。在某些情况下，可以使用这些扩展指数值集代替标准值集来表示类型的值 float或double。

任何浮点值集的有限非零值都可以用s · m · 2 （e - N + 1）的形式表示，其中s是+1或-1，m是小于2 N的正整数，和Ë是之间的整数ë 分钟 = - （2 ķ -1和-2）ë 最大 = 2 ķ -1 -1，包容，并且其中ñ和K是取决于值集的参数。有些值可以用多种方式表示在这种形式中; 例如，假设值集中的值 v可能使用 s， m和 e的某些值以此形式表示，那么如果发生 m为偶数且 e小于2 K-1，则可以减半 m并将 e增加1以产生相同值 v的第二表示。在这种形式的表示被称为归一化的，如果米 ≥ 2 Ñ -1 ; 否则该表示被称为非规范化。如果在设定的值的值不能在这样的方式来表示中号 ≥ 2 Ñ -1，则该值被认为是一个非标准化的值，因为它没有归一化表示。

[表1.3.2-A](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2-140-A" \o "表2.3.2-A。 浮点值设置参数)总结了对两个必需和两个可选浮点值集的参数N和K（以及导出参数E min 和E max） 的约束。

表1.3.2-A。浮点值设置参数

| 参数 | 浮动 | 浮扩展-指数 | 双 | 双扩展指数 |
| --- | --- | --- | --- | --- |
| ñ | 24 | 24 | 53 | 53 |
| ķ | 8 | ≥ 11 | 11 | ≥ 15 |
| E 最大 | +127 | ≥+ 1023 | +1023 | ≥ + 16383 |
| E 分钟 | -126 | ≤- 1022 | -1022 | ≤- 16382 |

如果实现支持一个或两个扩展指数值集，则对于每个支持的扩展指数值集，存在特定的依赖于实现的常量K，其值受[表1.3.2-A的](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2-140-A" \o "表2.3.2-A。 浮点值设置参数)约束; 该值K反过来决定了E min 和E max的值。

四个值集中的每一个不仅包括上面归于它的有限非零值，还包括正零，负零，正无穷大，负无穷大和NaN的五个值。

请注意，[表1.3.2-A](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2-140-A" \o "表2.3.2-A。 浮点值设置参数)中的约束 被设计为浮点值集的每个元素都必须也是float-extended-exponent值集，double值集和double-extended-exponent值的元素。组。同样，double值集的每个元素也必然是double-extended-exponent值集的元素。每个扩展指数值集的指数值范围都大于相应的标准值集，但没有更高的精度。

浮点值集的元素正是可以使用IEEE 754标准中定义的单浮点格式表示的值，除了只有一个NaN值（IEEE 754指定2 24 -2个不同的NaN值）。double值集的元素正是可以使用IEEE 754标准中定义的双浮点格式表示的值，除了只有一个NaN值（IEEE 754指定2 53）-2个不同的NaN值）。但请注意，此处定义的float-extended-exponent和double-extended-exponent值集的元素分别不对应于可使用IEEE 754单扩展和双扩展格式表示的值。本说明书中并不强制为浮点值集合的值，除非浮点值必须在所表示的特定表示class文件格式（[§4.4.4](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.4.4" \o "4.4.4。 CONSTANT_Integer_info和CONSTANT_Float_info结构)， [§4.4.5](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.4.5" \o "4.4.5。 CONSTANT_Long_info和CONSTANT_Double_info结构)）。

float，float-extended-exponent，double和double-extended-exponent值集不是类型。对于Java虚拟机的实现，使用float值集的元素来表示type的值总是正确的float; 但是，在某些上下文中可能允许实现使用float-extended-exponent值集的元素。类似地，对于实现来说，使用double值集的元素来表示type的值总是正确的double; 但是，在某些上下文中，实现可能允许使用双扩展指数值集的元素。

除NaNs外，浮点值集的值是有序的。当从最小到最大排列时，它们是负无穷大，负有限值，正负零，正有限值和正无穷大。

浮点正零和浮点负零比较相等，但还有其他操作可以区分它们; 例如，分割1.0通过0.0产生正无穷大，但除以1.0 通过-0.0产生负无穷大。

NaNs是无序的，因此false如果其操作数中的任何一个或两个都是NaN ，则数值比较和数值相等的测试具有值。特别是，false当且仅当值为NaN时，对值自身的数值相等性的测试具有值。true如果任一操作数是NaN，则对数值不等式的测试具有值。

### 1.3.3 returnAddress类型和值

该returnAddress类型是使用Java虚拟机的JSR，RET和jsr\_w指令（[第JSR](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.jsr" \o "JSR)，[§ RET](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.ret" \o "RET)， [第jsr\_w](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.jsr_w" \o "jsr_w)）。该returnAddress 类型的值是指向Java虚拟机指令的操作码的指针。与数字基元类型不同，该returnAddress类型不对应于任何Java编程语言类型，并且不能由正在运行的程序修改。

### 1.3.4 boolean类型

尽管Java虚拟机定义了一种 boolean类型，但它只为它提供了非常有限的支持。没有专门用于boolean 值操作的Java虚拟机指令。相反，Java编程语言中对boolean值进行操作的表达式将被编译为使用Java虚拟机int数据类型的值。

Java虚拟机直接支持boolean数组。它newarray指令（[第newarray](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.newarray" \o "newarray)），能够创建boolean 阵列。类型的阵列boolean被访问并且使用经修改的 byte阵列指令baload和bastore （[§ baload](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.baload" \o "baload)，[§ bastore](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.bastore" \o "bastore)）。

在Oracle的Java虚拟机实现中，booleanJava编程语言中的byte数组被编码为Java虚拟机数组，每个boolean元素使用8位 。

Java虚拟机boolean 使用1来表示true和0表示数组组件false。如果编程boolean器将Java编程语言值映射到Java虚拟机类型的值int，则编译器必须使用相同的编码。

## 1.4 参考类型和值

有三种reference 类型：类类型，数组类型和接口类型。它们的值分别是对动态创建的类实例，数组或类实例或实现接口的数组的引用。

数组类型由具有单个维度的组件类型组成 （其长度不是由类型给出的）。数组类型的组件类型本身可以是数组类型。如果从任何数组类型开始，考虑其组件类型，然后（如果它也是数组类型）该类型的组件类型，依此类推，最终必须达到不是数组类型的组件类型; 这称为数组类型的元素类型。数组类型的元素类型必须是基本类型，类类型或接口类型。

reference值也可以是专用空引用的，没有对象的引用，这将在这里通过来表示null。该null引用最初没有运行时类型，但可以转换为任何类型。reference类型的默认值是null。

该规范没有强制要求具体的值编码null。

## 1.5 运行时数据区

Java虚拟机定义了在程序执行期间使用的各种运行时数据区域。其中一些数据区域是在Java虚拟机启动时创建的，仅在Java虚拟机退出时销毁。其他数据区域是每个线程。线程数据区域是在线程退出时创建和销毁线程时创建的。

### 1.5.1 pc注册

Java虚拟机可以同时支持许多执行线程（JLS§17）。每个Java虚拟机线程都有自己的 pc（程序计数器）寄存器。在任何时候，每个Java虚拟机线程都在执行单个方法的代码，即该线程的当前方法（第[2.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6。 框架)）。如果不是该方法 native，则pc寄存器包含当前正在执行的Java虚拟机指令的地址。如果线程当前正在执行该方法native，则Java虚拟机pc 寄存器的值未定义。Java虚拟机pcregister足够宽，可以returnAddress在特定平台上保存或指向本机指针。

### 1.5.2 Java虚拟机堆栈

每个Java虚拟机线程都有一个私有Java虚拟机堆栈，与线程同时创建。Java虚拟机堆栈存储帧（第[2.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6。 框架)）。Java虚拟机堆栈类似于传统语言的堆栈，例如C：它保存局部变量和部分结果，并在方法调用和返回中起作用。由于除了推送和弹出帧之外，永远不会直接操作Java虚拟机堆栈，因此可以对堆进行堆分配。Java虚拟机堆栈的内存不需要是连续的。

在第一版中的Java ®虚拟机规范，Java虚拟机堆被称为Java堆栈。

此规范允许Java虚拟机堆栈具有固定大小或根据计算的需要动态扩展和收缩。如果Java虚拟机堆栈具有固定大小，则可以在创建该堆栈时独立选择每个Java虚拟机堆栈的大小。

Java虚拟机实现可以为程序员或用户提供对Java虚拟机堆栈的初始大小的控制，以及在动态扩展或收缩Java虚拟机堆栈的情况下，控制最大和最小大小。

以下异常条件与Java虚拟机堆栈相关联：

如果线程中的计算需要比允许的更大的Java虚拟机堆栈，则Java虚拟机会抛出一个StackOverflowError。

如果可以动态扩展Java虚拟机堆栈，并且尝试进行扩展但可以使内存不足以实现扩展，或者可以使内存不足以为新线程创建初始Java虚拟机堆栈，则Java Virtual机器抛出一个OutOfMemoryError。

### 1.5.3 堆

Java虚拟机具有在所有Java虚拟机线程之间共享的堆。堆是运行时数据区，从中分配所有类实例和数组的内存。

堆是在虚拟机启动时创建的。对象的堆存储由自动存储管理系统（称为垃圾收集器）回收 ; 对象永远不会被显式释放。Java虚拟机假设没有特定类型的自动存储管理系统，可以根据实现者的系统要求选择存储管理技术。堆可以具有固定大小，或者可以根据计算的需要进行扩展，并且如果不需要更大的堆，则可以收缩。堆的内存不需要是连续的。

Java虚拟机实现可以为程序员或用户提供对堆的初始大小的控制，以及如果可以动态扩展或收缩堆，则控制最大和最小堆大小。

以下异常情况与堆相关联：

如果计算需要的堆量超过自动存储管理系统可用的堆，则Java虚拟机会抛出一个 OutOfMemoryError。

### 1.5.4 方法区

Java虚拟机具有在所有Java虚拟机线程之间共享的方法区域。方法区域类似于传统语言的编译代码的存储区域或类似于操作系统进程中的“文本”段。它存储每类结构，例如运行时常量池，字段和方法数据，以及方法和构造函数的代码，包括类和实例初始化以及接口初始化中使用的特殊方法（第[2.9节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.9" \o "2.9。 特殊方法)）。

方法区域是在虚拟机启动时创建的。虽然方法区域在逻辑上是堆的一部分，但是简单的实现可能选择不垃圾收集或压缩它。本规范未规定方法区域的位置或用于管理编译代码的策略。方法区域可以是固定大小的，或者可以根据计算的需要进行扩展，并且如果不需要更大的方法区域，则可以缩小方法区域。方法区域的内存不需要是连续的。

Java虚拟机实现可以提供程序员或用户对方法区域的初始大小的控制，以及在变大小方法区域的情况下，控制最大和最小方法区域大小。

以下异常条件与方法区域相关联：

如果方法区域中的内存无法满足分配请求，则Java虚拟机会抛出一个OutOfMemoryError。

### 1.5.5 运行时常量池

甲运行时间常数池是的每个类或每个接口的运行时表示constant\_pool在表class文件（[§4.4](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.4" \o "4.4。 恒定池)）。它包含几种常量，从编译时已知的数字文字到必须在运行时解析的方法和字段引用。运行时常量池提供类似于传统编程语言的符号表的功能，尽管它包含比典型符号表更宽范围的数据。

每个运行时常量池都是从Java虚拟机的方法区域（第[2.5.4节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.5.4" \o "2.5.4。 方法区)）中分配的。当Java虚拟机创建类或接口（第[5.3节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html" \l "jvms-5.3" \o "5.3。 创作和加载)）时，将构造类或接口的运行时常量池。

以下异常条件与类或接口的运行时常量池的构造相关联：

在创建类或接口时，如果运行时常量池的构造需要的内存比Java虚拟机的方法区域中可用的内存多，则Java虚拟机会抛出一个OutOfMemoryError。

有关 构造运行时常量池的信息*[，](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html" \o "Chapter 5. Loading, Linking, and Initializing)*请参见*[§5（加载，链接和初始化）](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html" \o "第5章加载，链接和初始化)*。

### 1.5.6 本机方法堆栈

Java虚拟机的实现可以使用常规堆栈（俗称“C堆栈”）来支持native方法（用Java编程语言以外的语言编写的方法）。本机方法堆栈也可以通过以诸如C语言的Java虚拟机的指令集的解释器的实现来使用。无法加载native 方法并且本身不依赖于传统堆栈的Java虚拟机实现不需要提供本机方法堆栈。如果提供，则通常在创建每个线程时为每个线程分配本机方法堆栈。

此规范允许本机方法堆栈具有固定大小或根据计算的需要动态扩展和收缩。如果本机方法堆栈具有固定大小，则可以在创建该堆栈时独立地选择每个本机方法堆栈的大小。

Java虚拟机实现可以为程序员或用户提供对本机方法堆栈的初始大小的控制，以及在不同大小的本机方法堆栈的情况下，控制最大和最小方法堆栈大小。

以下异常条件与本机方法堆栈相关联：

如果线程中的计算需要比允许的更大的本机方法堆栈，则Java虚拟机会抛出一个StackOverflowError。

如果可以动态扩展本机方法堆栈并尝试进行本机方法堆栈扩展，但可以使内存不足，或者如果没有足够的内存可用于为新线程创建初始本机方法堆栈，则Java虚拟机会抛出OutOfMemoryError。

## 1.6 框架

甲帧 用于存储数据和部分结果，以及执行动态链接，对方法和调度异常返回值。

每次调用方法时都会创建一个新帧。当方法调用完成时，框架将被销毁，无论该完成是正常还是突然（它会抛出未捕获的异常）。帧是从创建帧的线程的Java虚拟机堆栈（第[2.5.2节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.5.2" \o "2.5.2。 Java虚拟机堆栈)）中分配的。每个帧都有自己的局部变量数组（第[2.6.1节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6.1" \o "2.6.1。 局部变量)），它自己的操作数堆栈（第[2.6.2节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6.2" \o "2.6.2。 操作数堆栈)），以及对当前方法类的运行时常量池（第[2.5.5节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.5.5" \o "2.5.5。 运行时常量池)）的引用。

可以使用附加的特定于实现的信息来扩展帧，例如调试信息。

局部变量数组和操作数堆栈的大小在编译时确定，并与与帧相关的方法的代码一起提供（第[4.7.3节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.7.3" \o "4.7.3。 代码属性)）。因此，帧数据结构的大小仅取决于Java虚拟机的实现，并且可以在方法调用上同时分配这些结构的存储器。

只有一个帧（执行方法的帧）在给定控制线程中的任何点处都是活动的。该帧被称为当前帧，并且其方法被称为当前方法。定义当前方法的类是当前类。局部变量和操作数堆栈的操作通常参考当前帧。

如果框架的方法调用另一个方法或其方法完成，则框架将不再是当前框架。调用方法时，会创建一个新帧，并在控制转移到新方法时变为当前帧。在方法返回时，当前帧将其方法调用的结果（如果有）传递回前一帧。然后当前一帧成为当前帧时丢弃当前帧。

请注意，由线程创建的帧对于该线程是本地的，并且不能被任何其他线程引用。

### 1.6.1局部变量

每个帧（第[2.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6。 框架)）包含一个称为局部变量的变量数组。帧的局部变量数组的长度在编译时确定，并以类或接口的二进制表示形式提供，同时提供与帧相关的方法的代码（第[4.7.3节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.7.3" \o "4.7.3。 代码属性)）。

单个本地变量可以保存类型的值boolean，byte，char，short，int， float，reference，或returnAddress。一对局部变量可以包含类型long或值double。

通过索引来解决局部变量。第一个局部变量的索引为零。当且仅当该整数在0到1之间且小于局部变量数组的大小时，整数才被认为是局部变量数组的索引。

类型long或类型的值double占用两个连续的局部变量。只能使用较小的索引来处理这样的值。例如，double存储在索引n的局部变量数组中的类型值 实际上占用索引为n和 n + 1 的局部变量; 但是，无法加载索引n +1 处的局部变量。它可以存储到。但是，这样做会使局部变量n的内容无效。

Java虚拟机不要求 n是偶数。直观地讲，的类型的值long而 double不必是局部变量阵列中的64位对齐的。实现者可以使用为该值保留的两个局部变量自由决定表示此类值的适当方法。

Java虚拟机使用局部变量在方法调用上传递参数。在类方法调用中，任何参数都在从局部变量0开始的连续局部变量中传递。在实例方法调用中，局部变量0始终用于传递对调用实例方法的对象的引用（this使用Java编程语言）。随后，任何参数都在从局部变量1开始的连续局部变量中传递。

### 1.6.2操作数堆栈

每个帧（第[2.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6。 框架)）包含一个后进先出（LIFO）堆栈，称为其操作数堆栈。帧的操作数堆栈的最大深度在编译时确定，并与用于与帧相关的方法的代码一起提供（第[4.7.3节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.7.3" \o "4.7.3。 代码属性)）。

在上下文清楚的地方，我们有时会将当前帧的操作数堆栈简称为操作数堆栈。

当创建包含它的帧时，操作数堆栈为空。Java虚拟机提供指令以将局部变量或字段中的常量或值加载到操作数堆栈上。其他Java虚拟机指令从操作数堆栈中获取操作数，对它们进行操作，并将结果推回操作数堆栈。操作数堆栈还用于准备要传递给方法和接收方法结果的参数。

例如，IADD 指令（[§ IADD](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.iadd" \o "我加)）将两个int值加在一起。它要求int要添加的值是操作数堆栈的前两个值，由前面的指令推送到那里。这两个int值都从操作数堆栈中弹出。它们被添加，它们的总和被推回到操作数堆栈上。子计算可以嵌套在操作数堆栈上，从而产生可以由包含计算使用的值。

操作数堆栈上的每个条目都可以包含任何Java虚拟机类型的值，包括类型long或类型的值 double。

必须以适合其类型的方式操作操作数堆栈中的值。例如，不可能推送两个int值，然后将它们视为a long或推送两个float值，然后使用iadd指令添加它们。少数的Java虚拟机指令（DUP指令（[§ DUP](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.dup" \o "DUP)）和交换（[§ 交换](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.swap" \o "交换)））将运行时数据区域作为原始值进行操作，而不考虑其特定类型; 这些指令的定义方式使它们不能用于修改或分解单个值。操作数堆栈操作的这些限制是通过class文件验证（第[4.10节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.10" \o "4.10。 验证类文件)）强制执行的。

在任何时间点，操作数堆栈具有相关联的深度，其中类型的值long或 double贡献两个单位的深度和任何其他类型的值贡献一个单位。

### 1.6.3动态链接

每个帧（第[2.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6。 框架)）包含对运行时常量池（第[2.5.5](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.5.5" \o "2.5.5。 运行时常量池)[节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6。 框架)）的引用，用于支持方法代码的动态链接的当前方法的类型。该class方法的文件代码是指要调用的方法和要通过符号引用访问的变量。动态链接将这些符号方法引用转换为具体的方法引用，根据需要加载类以解析尚未定义的符号，并将变量访问转换为与这些变量的运行时位置相关联的存储结构中的适当偏移。

方法和变量的这种后期绑定使得方法使用的其他类中的更改不太可能破坏此代码。

### 1.6.4正常方法调用完成

如果调用不会直接从Java虚拟机或执行显式语句引发异常（第[2.10节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.10" \o "2.10。 例外)）， 则方法调用 会正常完成。如果当前方法的调用正常完成，则可以将值返回给调用方法。当被调用的方法执行其中一个返回指令（第[2.18.8节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.11.8" \o "2.11.8。 方法调用和返回指令)）时，就会发生这种情况，返回指令的选择必须适合于返回值的类型（如果有的话）。 throw

在这种情况下，当前帧（第[2.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6。 框架)）用于恢复调用者的状态，包括其局部变量和操作数堆栈，调用者的程序计数器适当地递增以跳过方法调用指令。然后执行在调用方法的帧中正常继续，返回值（如果有）被推送到该帧的操作数堆栈。

### 1.6.5突然的方法调用完成

如果在方法中执行Java虚拟机指令导致Java虚拟机抛出异常（第[2.10节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.10" \o "2.10。 例外)），并且该异常未在该方法中处理，则方法调用会 突然完成。一个执行athrow指令（[§ athrow](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.athrow" \o "athrow)）也导致异常被抛出明确，如果该异常没有被当前的方法抓住了，结果突然的方法调用完成。突然完成的方法调用永远不会向其调用者返回值。

## 1.7 对象的表示

Java虚拟机不要求对象的任何特定内部结构。

在Oracle的一些Java虚拟机实现中，对类实例的引用是指向句柄的指针，该句柄本身是一对指针：一个指向包含对象方法的表和指向Class表示对象的对象的指针。 对象的类型，另一个是从堆中为对象数据分配的内存。

## 1.8浮点运算

Java虚拟机包含IEEE二进制浮点运算标准（ANSI / IEEE Std.754-1985，New York）中规定的浮点运算的子集。

### 1.8.1 Java虚拟机浮点运算和IEEE 754

Java虚拟机支持的浮点运算与IEEE 754标准之间的主要区别是：

Java虚拟机的浮点操作不会抛出异常，陷阱或以其他方式发出IEEE 754无效操作异常条件，除零，溢出，下溢或不精确的信号。Java虚拟机没有信令NaN值。

Java虚拟机不支持IEEE 754信令浮点比较。

Java虚拟机的舍入操作始终使用IEEE 754舍入到最近模式。不精确的结果四舍五入到最接近的可表示值，并且连接到具有零最低有效位的值。这是IEEE 754默认模式。但Java虚拟机指令将浮点类型的值转换为整数类型的值，舍入为零。Java虚拟机没有提供任何改变浮点舍入模式的方法。

Java虚拟机不支持IEEE 754单扩展或双扩展格式，除非可以说double和double-extended-exponent值集支持单扩展格式。float-extended-exponent和double-extended-exponent值集（可选择支持）与IEEE 754扩展格式的值不对应：IEEE 754扩展格式需要扩展精度以及扩展指数范围。

### 1.8.2 浮点模式

每个方法都有一个 浮点模式，它是FP-strict或FP-strict。方法的浮点模式由定义方法的结构项（第[4.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.6" \o "4.6。 方法)）的ACC\_STRICT标志 的设置确定。设置该标志的方法是FP严格的; 否则，该方法不是FP严格的。 access\_flagsmethod\_info

请注意，该ACC\_STRICT标志的映射意味着由JDK 1.1版或更早版本中的编译器编译的类中的方法实际上不是FP严格的。

当调用创建包含操作数堆栈的帧的方法具有浮点模式时，我们将操作数堆栈称为具有给定的浮点模式。类似地，当包含该指令的方法具有该浮点模式时，我们将Java虚拟机指令称为具有给定的浮点模式。

如果支持float-extended-exponent值集（第[2.3.2节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2" \o "2.3.2。 浮点类型，值集和值)），float则非FP-strict的操作数堆栈上的类型值 可以超出该值集，除非值集转换禁止（[§2.8.3](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.8.3" \o "2.8.3。 价值集转换)） 。如果支持双扩展指数值集（第[2.3.2节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2" \o "2.3.2。 浮点类型，值集和值)），double则非FP严格的操作数堆栈上的类型值 可以超出该值集，除非值集转换禁止。

在所有其他情况下，无论操作数堆栈上或其它地方，也不管浮点模式，键入的浮点值float以及 double可以仅在范围内浮动值设置和双值集合，分别。特别是，类和实例字段，数组元素，局部变量和方法参数只能包含从标准值集中提取的值。

### 1.8.3价值集转换

在特定情况下，允许或要求支持扩展浮点值集的Java虚拟机的实现，以在扩展值集和标准值集之间映射关联浮点类型的值。这样的值集转换不是类型转换，而是与相同类型关联的值集之间的映射。

在指示值集转换的情况下，允许实现对值执行以下操作之一：

如果值是类型float且不是浮点值集的元素，则它将值映射到浮点值集的最近元素。

如果值是类型double且不是double值集的元素，则它将值映射到double值集的最近元素。

此外，在指示值集转换的情况下，需要执行某些操作：

假设执行非FP-strict的Java虚拟机指令会导致类型的值 float被推送到FP-strict的操作数堆栈，作为参数传递，或存储到局部变量，字段或元素中一个数组。如果该值不是浮点值集的元素，则将该值映射到浮点值集的最近元素。

假设执行非FP-strict的Java虚拟机指令会导致类型的值 double被推送到FP-strict的操作数堆栈，作为参数传递，或存储到局部变量，字段或元素中一个数组。如果该值不是double值集的元素，则将值映射到double值集的最近元素。

在方法调用期间传递浮点类型的参数（包括native 方法调用）可能会发生这种所需的值集转换; 将浮点类型的值从非FP-strict的方法返回到FP-strict的方法; 或者将浮点类型的值存储到非FP严格的方法中的局部变量，字段或数组中。

并非扩展指数值集中的所有值都可以精确映射到相应标准值集中的值。如果映射的值太大而无法准确表示（其指数大于标准值集允许的值），则将其转换为相应类型的（正或负）无穷大。如果映射的值太小而不能精确表示（其指数小于标准值集所允许的值），则将其舍入到最接近的可表示的非规范化值或相同符号的零。

值集转换会保留无穷大和NaN，并且不能更改正在转换的值的符号。值集转换对不是浮点类型的值没有影响。

## 1.9特殊方法

在Java虚拟机级别，使用Java编程语言（JLS§8.8）编写的每个构造函数都显示为具有特殊名称的实例初始化方法<init>。该名称由编译器提供。因为名称<init>不是有效的标识符，所以它不能直接用在用Java编程语言编写的程序中。实例的初始化方法可能仅在由Java虚拟机调用invokespecial指令（[第invokespecial](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.invokespecial" \o "invokespecial)），它们只能在未初始化的类实例上调用。实例初始化方法采用从中派生的构造函数的访问权限（JLS§6.6）。

类或接口最多只有一个类或接口初始化方法， 并通过调用该方法进行初始化（第[5.5节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html" \l "jvms-5.5" \o "5.5。 初始化)）。类或接口的初始化方法具有特殊名称<clinit>，不带参数，并且是void（第[4.3.3节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.3.3" \o "4.3.3。 方法描述符)）。

命名其他方法<clinit>在class文件中都没有结果。它们不是类或接口初始化方法。它们不能被任何Java虚拟机指令调用，并且永远不会被Java虚拟机本身调用。

在class版本号为51.0或更高版本的文件中，该方法必须另外设置其 ACC\_STATIC标志（第[4.6节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.6" \o "4.6。 方法)），以便成为类或接口初始化方法。

此要求是在Java SE 7中引入的。在版本号为50.0或更低版本的类文件中，名为 <clinit>void且不带参数的方法被认为是类或接口初始化方法，无论其ACC\_STATIC标志的设置如何 。

该名称<clinit>由编译器提供。因为名称<clinit>不是有效的标识符，所以它不能直接用在用Java编程语言编写的程序中。Java虚拟机隐式调用类和接口初始化方法; 它们永远不会直接从任何Java虚拟机指令调用，但只能作为类初始化过程的一部分间接调用。

如果满足以下所有条件，则方法是签名多态的：

它在java.lang.invoke.MethodHandle课堂上宣布。

它有一个类型的形式参数 Object[]。

它的返回类型为Object。

它有ACC\_VARARGS和ACC\_NATIVE标志设置。

在Java SE 8中，唯一的签名多态方法是类的方法invoke和invokeExact方法 java.lang.invoke.MethodHandle。

Java虚拟机提供了特殊处理，以在签名多态性方法invokevirtual 指令（[第invokevirtual](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.invokevirtual" \o "invokevirtual)），以实现一个的调用方法处理。方法句柄是对基础方法，构造函数，字段或类似的低级操作（第[5.4.3.5节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html" \l "jvms-5.4.3.5" \o "5.4.3.5。 方法类型和方法句柄解析)）的强类型，可直接执行的引用，具有可选的参数或返回值转换。这些转换非常通用，包括转换，插入，删除和替换等模式。见java.lang.invoke 有关更多信息，请参阅Java SE平台API中的包。

## 1.10例外

Java虚拟机中的异常由类的实例Throwable或其子类之一表示。抛出异常会导致控制从抛出异常的位置立即进行非本地转移。

大多数异常由于它们发生的线程的动作而同步发生。相反，异步异常可能在程序执行的任何时刻发生。Java虚拟机因以下三个原因之一抛出异常：

一个athrow指令（[§ athrow](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.athrow" \o "athrow)）被执行。

Java虚拟机同步检测到异常执行条件。这些异常不是在程序中的任意点抛出，而是仅在执行以下指令之后同步：

将异常指定为可能的结果，例如：

当指令包含违反Java编程语言语义的操作时，例如在数组边界之外进行索引。

在加载或链接程序的一部分时发生错误。

导致超出资源的某些限制，例如，当使用太多内存时。

发生异步异常是因为：

stop类Thread 或被ThreadGroup调用的方法，或

Java虚拟机实现中发生内部错误。

stop一个线程可以调用这些方法来影响另一个线程或指定线程组中的所有线程。它们是异步的，因为它们可能在执行其他线程或线程的任何时刻发生。内部错误被视为异步（第[6.3节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.3" \o "6.3。 虚拟机错误)）。

Java虚拟机可能允许在抛出异步异常之前发生少量但有限的执行量。允许这种延迟允许优化代码在遵循Java编程语言的语义的同时检测并抛出这些异常。

一个简单的实现可能会在每个控制传输指令的点处轮询异步异常。由于程序具有有限大小，因此这提供了检测异步异常的总延迟的界限。由于控制传输之间不会发生异步异常，因此代码生成器可以灵活地在控制传输之间重新排序计算以获得更高的性能。本文有效地轮询对股票硬件马克·菲利，PROC。1993年丹麦哥本哈根功能编程和计算机体系结构会议，第179-187页，建议作为进一步阅读。

Java虚拟机抛出的异常是精确的：当发生控制转移时，必须在抛出异常的点之前执行的指令的所有效果都必须发生。在抛出异常的点之后没有出现的指令似乎已被评估。如果优化代码推测性地执行了一些遵循发生异常的点的指令，则必须准备这样的代码以将该推测执行隐藏在程序的用户可见状态中。

Java虚拟机中的每个方法可以与零个或多个异常处理程序相关联。异常处理程序指定实现异常处理程序处于活动状态的方法的Java虚拟机代码的偏移范围，描述异常处理程序能够处理的异常类型，并指定要处理的代码的位置那个例外。如果导致异常的指令的偏移量在异常处理程序的偏移范围内，并且异常类型与异常处理程序处理的异常类的子类相同，则异常与异常处理程序匹配。抛出异常时，Java虚拟机会在当前方法中搜索匹配的异常处理程序。

如果在当前方法中找不到这样的异常处理程序，则当前方法调用突然完成（第[2.6.5节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6.5" \o "2.6.5。 突然的方法调用完成)）。在突然完成时，将丢弃当前方法调用的操作数堆栈和局部变量，并弹出其框架，恢复调用方法的框架。然后在调用者的框架的上下文中重新抛出该异常，依此类推，继续向上调用方法调用链。如果在到达方法调用链的顶部之前未找到合适的异常处理程序，则终止执行抛出异常的线程。

搜索方法的异常处理程序以进行匹配的顺序非常重要。在class文件中，每个方法的异常处理程序都存储在一个表中（第[4.7.3节](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.7.3" \o "4.7.3。 代码属性)）。在运行时，当抛出异常时，Java虚拟机将按照它们出现在class文件中相应异常处理程序表中的顺序搜索当前方法的异常处理程序，从该表的开头开始。

请注意，Java虚拟机不会强制执行方法的异常表条目的嵌套或任何排序。Java编程语言的异常处理语义只能通过与编译器的合作来实现（[§3.12](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.html" \l "jvms-3.12" \o "3.12。 投掷和处理例外情况)）。当class通过其他方式生成文件时，定义的搜索过程可确保所有Java虚拟机实现的行为始终如一。

# 英文原文

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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.11.1" \o "2.11.1. Types and the Java Virtual Machine).

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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.4" \o "2.3.4. The boolean Type)), and the returnAddress type ([§2.3.3](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.3" \o "2.3.3. The returnAddress Type and Values)).

The numeric types consist of the integral types ([§2.3.1](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.1" \o "2.3.1. Integral Types and Values)) and the floating-point types ([§2.3.2](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2" \o "2.3.2. Floating-Point Types, Value Sets, and Values)).

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 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. The Second Edition of The Java® Virtual Machine Specification 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 (-27 to 27 - 1), inclusive

For short, from -32768 to 32767 (-215 to 215 - 1), inclusive

For int, from -2147483648 to 2147483647 (-231 to 231 - 1), inclusive

For long, from -9223372036854775808 to 9223372036854775807 (-263 to 263 - 1), inclusive

For char, from 0 to 65535 inclusive

### 2.3.2 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, 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 setand 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 ⋅ m ⋅ 2(e − N + 1), where s is +1 or −1, m is a positive integer less than 2N, and e is an integer between Emin = −(2K−1−2) and Emax = 2K−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 2K-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 ≥ 2N-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 ≥ 2N-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 2.3.2-A](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2-140-A" \o "Table 2.3.2-A. Floating-point value set parameters).

Table 2.3.2-A. 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 2.3.2-A](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2-140-A" \o "Table 2.3.2-A. Floating-point value set parameters); 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 2.3.2-A](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.3.2-140-A" \o "Table 2.3.2-A. Floating-point value set parameters) 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 224-2 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 253-2 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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.4.4" \o "4.4.4. The CONSTANT_Integer_info and CONSTANT_Float_info Structures), [§4.4.5](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.4.5" \o "4.4.5. The CONSTANT_Long_info and CONSTANT_Double_info Structures)).

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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.jsr" \o "jsr), [§ret](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.ret" \o "ret), [§jsr\_w](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.jsr_w" \o "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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.newarray" \o "newarray)) enables creation of boolean arrays. Arrays of type boolean are accessed and modified using the byte array instructions baload and bastore ([§baload](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.baload" \o "baload), [§bastore](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html" \l "jvms-6.5.bastore" \o "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 booleanelement.

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.

This 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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6. Frames)) 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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.6" \o "2.6. Frames)). 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 First Edition of The Java® Virtual Machine Specification, 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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.9" \o "2.9. Special Methods)) 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 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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html" \l "jvms-4.4" \o "4.4. The Constant Pool)). 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](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html" \l "jvms-2.5.4" \o "2.5.4. Method Area)). The run-time constant pool for a class or interface is constructed when the class or interface is created ([§5.3](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html" \l "jvms-5.3" \o "5.3. Creation and Loading)) 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 (Loading, Linking, and Initializing)](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html" \o "Chapter 5. Loading, Linking, and Initializing)* 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 run-time 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

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).

### 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 was introduced 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 all of the following are true:

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 8, 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 strongly 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.