Jvm的原始类型：

数值类型（int，float，double）、 布尔类型 （Boolean）和 returnAddress类型三类

returnAddress类型：被 Java 虚拟机的 jsr、ret 和 jsr\_w 指令所使用。returnAddress 类型的值指向一条虚拟机指令的操作码，returnAddress 类型在 Java 语言之中并不存在相应的类型，也无法在程序运行期间更改returnAddress 类型的值。

Boolean在jvm中是转化为int类型进行操作的。

Java 虚拟机中有三种引用类型：类类型（Class Types）、数组类型（Array Types）和接口类型（Interface Types）

这些引用类型的值分别由类实例、数组实例和实现了某个接口的类实例或数组实例动态创建。

Null类型的引用可以指向任意类型的对象

**PC 寄存器**

Java 虚拟机可以支持多条线程同时执行，每一条 Java虚拟机线程都有自己的 PC（Program Counter）寄存器。在任意时刻，一条 Java 虚拟机线程只会执行一个方法的代码，这个正在被线程执行的方法称为该线程的当前方法（Current Method。如果这个方法不是 native 的，那 PC 寄存器就保存 Java 虚拟机正在执行的字节码指令的地址，如果该方法是 native 的，那 PC 寄存器的值是 undefined。 PC 寄存器的容量至少应当能保存一个 returnAddress 类型的数据或者一个与平台相关的本地指针的值。

**Java 虚拟机栈**

每一条 Java 虚拟机线程都有自己私有的 Java 虚拟机栈（Java Virtual Machine Stack）。这个栈与线程同时创建，用于存储栈帧（Frames）。Java 虚拟机栈的作用与传统语言（例如 C 语言）中的栈非常类似，就是用于存储局部变量与一些过程结果的地方另外，它在方法调用和返回中也扮演了很重要的角色。因为除了栈帧的出栈和入栈之外，Java 虚拟机栈不会再受其他因素的影响，所以栈帧可以在堆中分配②，Java 虚拟机栈所使用的内存不需要保证是连续的。

**Java 堆**

在 Java 虚拟机中，堆（Heap）是可供各条线程共享的运行时内存区域，也是供所有类实例和数组对象分配内存的区域。 Java 堆在虚拟机启动的时候就被创建，它存储了被自动内存管理系统（Automatic Storage Management System，也即是常说的“Garbage Collector（垃圾收集器GC）” ）所管理的各种对象，这些受管理的对象无需，也无法显式地被销毁。

**方法区**

在 Java 虚拟机中，方法区（Method Area）是可供各条线程共享的运行时内存区域。方法区与传统语言中的编译代码储存区（Storage Area Of Compiled Code）或者操作系统进程的正文段（Text Segment）的作用非常类似，它存储了每一个类的结构信息，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容、还包括一些在类、实例、接口初始化时用到的特殊方法。

**运行时常量池**

运行时常量池（Runtime Constant Pool）是每一个类或接口的常量池（Constant\_Pool）的运行时表示形式，它包括了若干种不同的常量：从编译期可知的数值字面量到必须运行期解析后才能获得的方法或字段引用。运行时常量池扮演了类似传统语言中符号表（Symbol Table）的角色，不过它存储数据范围比通常意义上的符号表要更为广泛。每一个运行时常量池都分配在 Java 虚拟机的方法区之中（static部分），在类和接口被加载到虚拟机后（class对象创建的时候），对应的运行时常量池就被创建出来。

**本地方法栈**

Java 虚拟机实现可能会使用到传统的栈（通常称之为“C Stacks”）来支持 native 方法（指使用 Java 以外的其他语言编写的方法）的执行，这个栈就是本地方法栈（Native Method Stack）。

**栈帧**

栈帧（Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接（Dynamic Linking）、方法返回值和异常分派（Dispatch Exception）。栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法结束。栈帧的存储空间分配在 Java 虚拟机栈之中，每一个栈帧都有自己的局部变量表（Local Variables）、操作数栈（Operand Stack，）和指向当前方法所属的类的运行时常量池的引用。

**局部变量表（存储结构）**

每个栈帧内部都包含一组称为局部变量表（Local Variables）的变量列表。栈帧中局部变量表的长度由编译期决定，并且存储于类和接口的二进制表示之中，既通过方法的Code 属性保存及提供给栈帧使用。Java 虚拟机使用局部变量表来完成方法调用时的参数传递，当一个方法被调用的时候，它的参数将会传递至从 0 开始的连续的局部变量表位置上。特别地，当一个实例方法被调用的时候，第 0 个局部变量一定是用来存储被调用的实例方法所在的对象的引用（即 Java 语言中的“this”关键字）。后续的其他参数将会传递至从 1 开始的连续的局部变量表位置上。

**操作数栈（程序运行时的逻辑结构）**

每一个栈帧（§2.6）内部都包含一个称为操作数栈（Operand Stack）的后进先出（Last-In-First-Out，LIFO）栈。栈帧中操作数栈的长度由编译期决定，并且存储于类和接口的二进制表示之中，既通过方法的 Code 属性保存及提供给栈帧使用。在上下文明确，不会产生误解的前提下，我们经常把“当前栈帧的操作数栈”直接简称为“操作数栈”。操作数栈所属的栈帧在刚刚被创建的时候，操作数栈是空的。Java 虚拟机提供一些字节码指令来从局部变量表或者对象实例的字段中复制常量或变量值到操作数栈中，也提供了一些指令用于从操作数栈取走数据、操作数据和把操作结果重新入栈。在方法调用的时候，操作数栈也用来准备调用方法的参数以及接收方法返回结果。

**动态链接**

每一个栈帧（§2.6）内部都包含一个指向运行时常量池（§2.5.5）的引用来支持当前方法的代码实现动态链接 （Dynamic Linking）。在 Class 文件里面，描述一个方法调用了其他方法，或者访问其成员变量是通过符号引用（Symbolic Reference）来表示的，动态链接的作用就是将这些符号引用所表示的方法转换为实际方法的直接引用。类加载的过程中将要解析掉尚未被解析的符号引用，并且将变量访问转化为访问这些变量的存储结构所在的运行时内存位置的正确偏移量。由于动态链接的存在，通过晚期绑定（Late Binding）使用的其他类的方法和变量在发生变化时，将不会对调用它们的方法构成影响。

初始化方法的特殊命名（构造函数经过编译器编程字节码中的init方法）

在 Java 虚拟机层面上，Java 语言中的构造函数在《Java 语言规范 （第三版）》 （下文简称JLS3，§8.8）是以一个名为<init>的特殊实例初始化方法的形式出现的，<init>这个方法名称是由编译器命名的，因为它并非一个合法的 Java 方法名字，不可能通过程序编码的方式实现。实例初始化方法只能在实例的初始化期间，通过 Java 虚拟机的 invokespecial 指令来调用，只有在实例正在构造的时候，实例初始化方法才可以被调用访问（JLS3，§6.6）。

异常

抛异常的本质实际上是程序控制权的一种即时的、非局部（Nonlocal）的转换——从异常抛出的地方转换至处理异常的地方。

Java 虚拟机中异常的出现总是由下面三种原因之一导致的：

1. 虚拟机同步检测到程序发生了非正常的执行情况，这时异常将会紧接着在发生非正常执行情况的字节码指令之后抛出。

例如：

 字节码指令所蕴含的操作违反了 Java 语言的语义，如访问一个超出数组边界范围的元素。

 类在加载或者链接时出现错误。

 使用某些资源的时候产生资源限制，例如使用了太多的内存。

2. athrow 字节码指令被执行。

当异常抛出时，Java 虚拟机搜索当前方法的包含的各个异常处理器，如果能找到可以处理该异常的异常处理器，则将代码控制权转向到异常处理器中描述的处理异常的分支之中。搜索异常处理器时的搜索顺序是很关键的，在 Class 文件里面，每个方法的异常处理器都存储在一个表中（§4.7.3）。在运行时，当有异常出现之后，Java 虚拟机就按照 Class 文件中的异常处理器表描述异常处理器的先后顺序，从前至后进行搜索。Java 虚拟机本身不会对方法的对异常处理器表做排序或者其他方式的强制处理。