JVM

# The Structure of the Java Virtual Machine

the Java Virtual Machine operates on two kinds of types: *primitive types* and *reference types*.

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

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

There are three kinds of reference types: class types, array types, and interface types.

## 2.5 Run-Time Data Areas

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

### 2.5.1 The pc Register

The Java Virtual Machine can support many threads of execution at once (JLS §17). Each Java Virtual Machine thread has its own pc (program counter) register. At any point, each Java Virtual Machine thread is executing the code of a single method, namely the current method (§2.6) for that thread. If that method is not native, the pc register contains the address of the Java Virtual Machine instruction currently being executed. If the method currently being executed by the thread is native, the value of the Java Virtual Machine's pc register is undefined. The Java Virtual Machine's pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.

### 2.5.2 Java Virtual Machine Stacks

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

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.

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.

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 used in class and interface initialization and in instance initialization (§2.9).

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.

The following exceptional condition is associated with the method area:  
• If memory in the method area cannot be made available to satisfy an allocation

request, the Java Virtual Machine throws an OutOfMemoryError.

### 2.5.5 Run-Time Constant Pool

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

Each run-time constant pool is allocated from the Java Virtual Machine's method area (§2.5.4). The run-time constant pool for a class or interface is constructed when the class or interface is created (§5.3) by the Java Virtual Machine.

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

• When creating a class or interface, if the construction of the run-time constant pool requires more memory than can be made available in the method area of the Java Virtual Machine, the Java Virtual Machine throws an OutOfMemoryError.

### 2.5.6 Native Method Stacks

Support Native Method

## 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.

### 2.6.1 Local Variables

Each frame (§2.6) contains an array of variables known as its *local variables*. 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.

A value of type long or type double occupies two consecutive local variables. Such a value may only be addressed using the lesser index.

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 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.

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.

## 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.