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 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 dynamically 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`.

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 `OutOfMemoryError`.

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 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 be 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