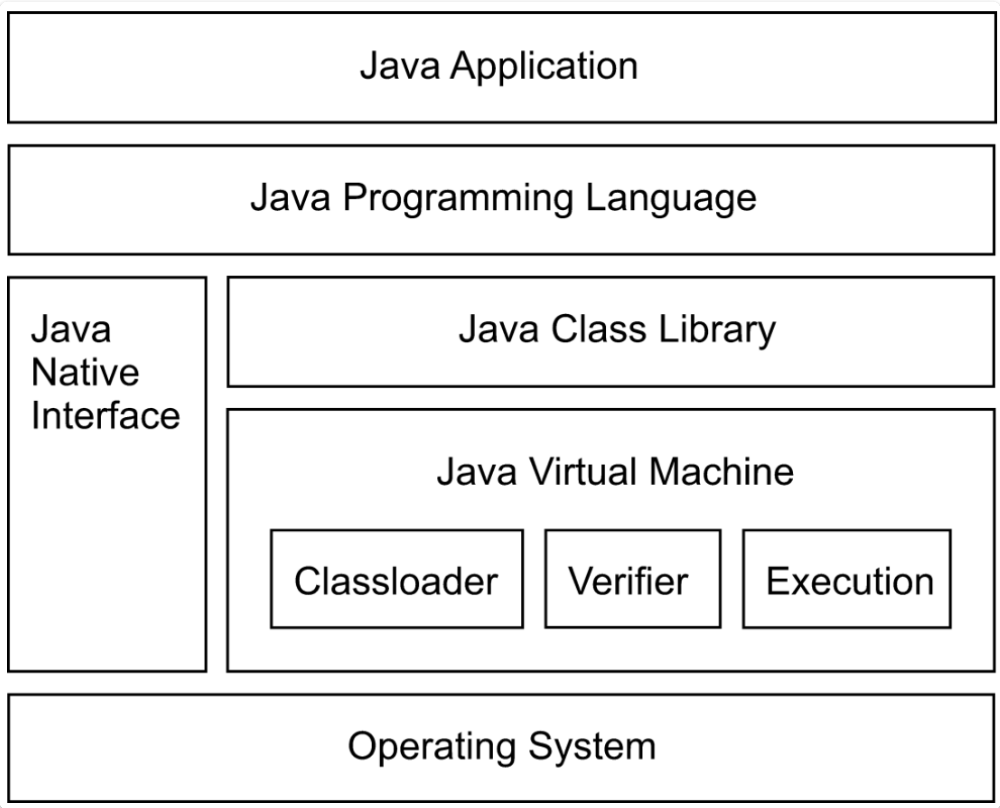
03 - Java Virtual Machine

Is a multi-threaded stack based machine: each instruction take arguments from top of the operand stack and also put results on top of operand stack (LIFO).

It defines the results of the compilation in bytecode (JVM's language) as a machine independent class file format (.class), that must be supported by all JVM implementations. It is abstract, so no implementation details like memory layout of run-time data area, garbage-collection, optimizations... Also object representation is left to the implementation, even the null value.

Note that *.class* is platform independent, but the loading process transforms .class to an internal representation, which is implementation dependent.

Java Hierarchy



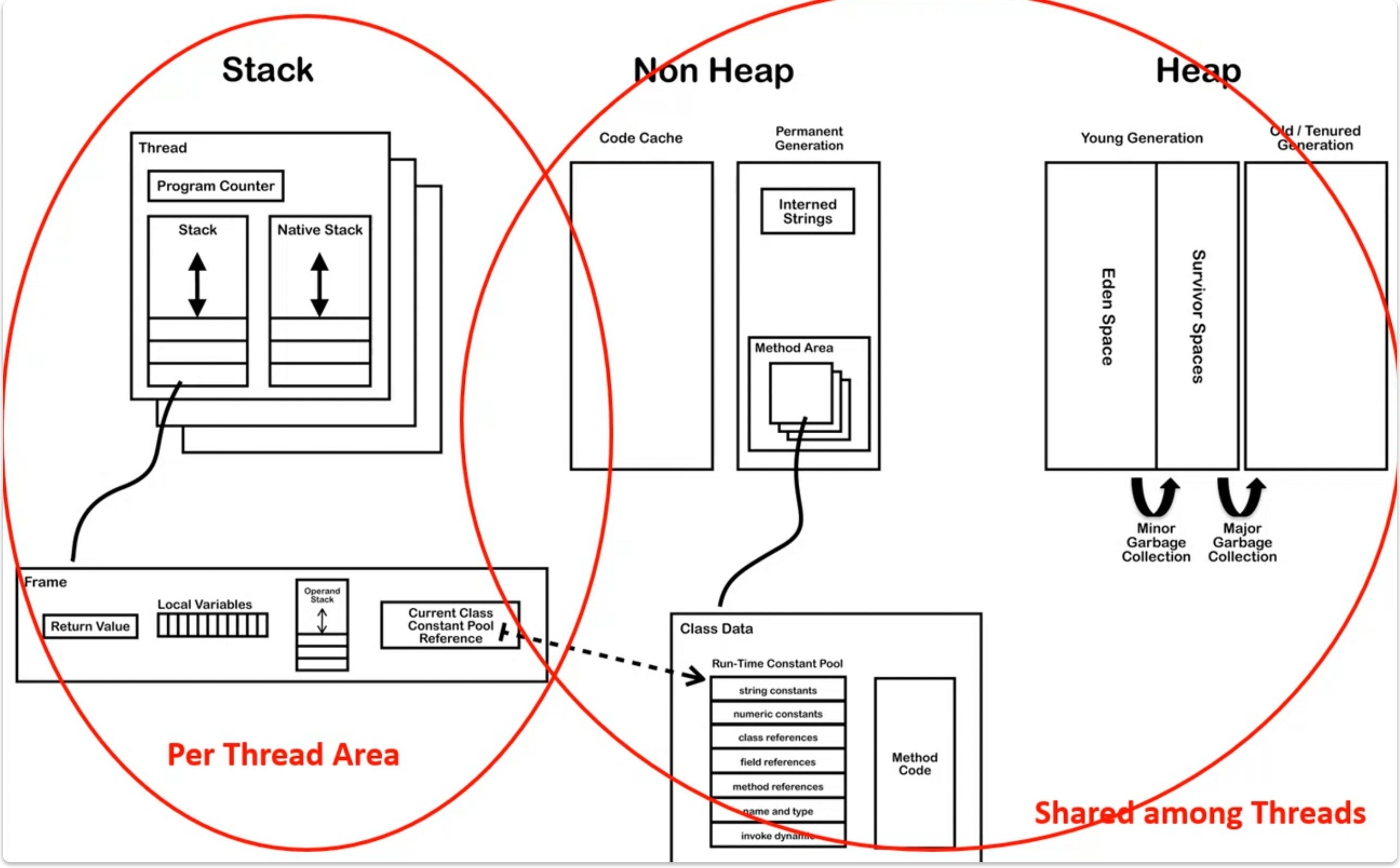
Thread

Multiple threads allowed per application (e.g. `main()` method), managed by time-slicing or parallelization. There is a direct mapping between a Java Thread and a native operating system Thread.

Instance of class `Thread`, started invoking `start()`. States: ready, in execution, suspended (`Thread.sleep()`), blocked (I/O or monitor request), waiting, terminated.

Daemon: background system threads (e.g. garbage collection, finalization, signal dispatching, ecc...).

Data Areas



- Per Thread:
  - PC (Program Counter);
  - Java stack (activation records -> method invocation/completion);
  - Native stack (Java Native Interface -> e.g. C function is invoked). Note that the frames in the stacks are not necessarily allocated contiguously, it could be implemented as a linked list (remember that JVM is an abstract machine);
- Shared among Threads (need thread safety):
  - Heap: objects and arrays (unlike C++). There is no explicit deallocation, the content will be garbage collected;
  - Non Heap: static part of the heap not garbage collected. Contains:
    - Code cache (JIT compilation): since interpreting bytecode is not as fast as native code, JVM look for bytecode executed several time and compile it to native code, stored in code cache;
    - Permanent generation (objects never deallocated):
      - Interned strings (string interning is a method of storing only one copy of each distinct string value, which must be immutable);
      - [Method Area](#) (memory where class files are loaded);

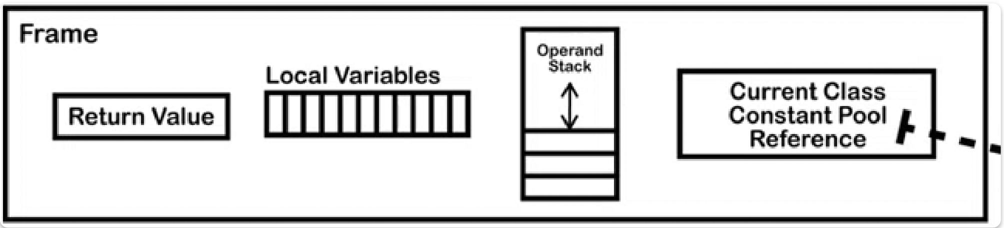
Example of class file:

ClassFile {

u4 magic;	**0xCAFEBABE**
u2 minor_version; u2 major_version;	**Java Language Version**
u2 constant_pool_count; cp_info contant_pool[constant_pool_count–1];	**Constant Pool**
u2 access_flags;	**access modifiers and other info**
u2 this_class; u2 super_class;	**References to Class and Superclass**
u2 interfaces_count; u2 interfaces[interfaces_count];	**References to Direct Interfaces**
u2 fields_count; field_info fields[fields_count];	**Static and Instance Variables**
u2 methods_count; method_info methods[methods_count];	**Methods**
u2 attributes_count; attribute_info attributes[attributes_count];	**Other Info on the Class**

}

Frame



- Local Variables Array:
  - Reference to this;
  - Method parameters;
  - Local variables.
- Operand Stack: expressions/methods evaluation;
- Reference to Constant Pool;
- Return value.

When a Java class is compiled, all the constants and all references to variables and methods are stored in the class constant pool as symbolic references. JVM can then resolve (bind) symbolic references in two ways:

- Eager (static): after class file is loaded;
- Lazy (late): when the reference is used for the first time.

This process is only done once, because the symbolic reference is completely replaced in the constant pool.

Method Area

It is the memory where class files are loaded, composed of:

- Classloader Reference;
- From the class file:
  - Run Time Constant Pool: constants / references (to fields/methods...);
  - Field data;
  - Method data;
    - Method code (bytecode);
    - ...

Classloader

All classes that are loaded contain a reference to the classloader that loaded them. In turn the classloader also contains a reference to all classes that it has loaded.

Generally works in 3 steps:

- Loading: read the class file, check if it is properly formatted and all data are recognized by JVM and create an appropriate class/interface (from bytecode to the internal representation specific to the JVM implementation);
- Linking: take the previously created class/interface and combine it with the run-time state of the JVM so that it can be executed. It consists in 3 steps:
  - Verification: overflow/underflow, types validity...;
  - Preparation: allocation of storage (method tables);
  - Resolution (optional, because of eager/lazy binding): resolve symbol references by loading referred classes/interfaces.
- Initialization: execute the class/interface method `clinit()` (initialize class variables). Instead, when instances are initialized, the `init()` method is called.

We have 3 main types:

- Bootstrap Class loader: written in native code, loads basic Java APIs (e.g.: rt.jar). During the JVM start-up, it loads the initial class;
- Extension Classloader: middle-layer loader, loads classes from standard Java extension APIs (e.g.: security extension functions);
- System Classloader: default application classloader, loads application classes from classpath.

Classloaders can also be user-defined, e.g.: runtime reloading of classes, loading from different sources (network, encrypted file...).

JVM Start-up and shutdown

This is the classic procedure when a program starts:

- The JVM starts up by loading an initial class (where the main method is) using the *bootstrap class loader*;
- The class is linked and initialized;
- The `public static void main(String[] args)` method is invoked;
- This will trigger loading, linking and initialization of additional classes and interfaces.

JVM exits when:

- all non-daemon threads terminate;
- `Runtime.exit`, `System.exit` (assuming it is secure).

Data Types

- Primitives: byte, short, int, long, char, float, double, boolean (supported only for arrays, since the smallest indexable word is of 1 byte and thus we can't read 1 bit, as the bool is), returnAddress (exception handling);
- References: class, array, interface;
- Opcodes: iadd, fadd... (types of operands).

However, there is no type information on local variables at runtime.

Instruction Set

Basic instructions that are executable on the virtual machine. They are the analogue of the instructions in machine language.

Properties:

- Variable length: one-byte opcode followed by arguments, against the stack machine of JVM (32 bit);
- Only relative branches (no GOTOs, jumps are done by only summing/subtracting values to the PC);
- Byte aligned: a data is x-byte aligned if it has a number of bytes which is multiple of x. The instruction set is 1-byte aligned, saving a lot of space but in order to read everything we must read 1 byte at a time;
- May contain symbolic references;

Examples:

- Load and store move back and forth data between the operand stack (where the stack machine JVM operates) and the local variable array;
- Arithmetic, control transfer, method invocation and return...

Moreover, each instruction could have different forms (like different constructors), e.g. different forms of "iload".

There are 3 addressing modes:

- Intermediate, a constant is part of instruction (e.g. iload_1: we know that we refer to the element in position 1 of the LVA);
- Indexed, access variable from local variable array;
- Stack, pop operand stack.

OpCodes are explicitly typed: the first letter of the OpCode identify the type. However, non-supported types will have to be converted, almost no support for *byte*, *char* and *short*. Instead, *int* is used as "computational type":

- i: int;
- l: long;
- s: short;
- b: byte;
- c: char;
- f: float;
- d: double;
- a: for reference.

Commands

- Compiler: `javac filename.java`;
- Disassembler: `javap -c -v filename.class` (option -c: produce entire bytecode, constant poll included);
- Execution: `java filename`.