James Power

The Java Virtual Machine

- JVM runtime behaviour
- JVM architecture
- .class file format
- JVM instruction set

Main source:

- The Java^{TM} Virtual Machine Specification (2^{nd} Ed)
  by Tim Lindholm & Frank Yellin
  Addison-Wesley, 1999

- `http://java.sun.com/docs/books/vmspec/`

JVM Runtime Behaviour

- VM startup
- Class Loading/Linking/Initialisation
- Instance Creation/Finalisation
- Unloading Classes
- VM exit

VM Startup and Exit

Startup

- Load, link, initialise class containing `main()`
- Invoke `main()` passing it the command-line arguments
- Exit when:
  - all non-daemon threads end, or
  - some thread explicitly calls `exit()` method

Class Loading

- Find the binary code for a class and create a corresponding `Class` object
- Done by a *class loader* - bootstrap, or create your own
- Optimise: prefetching, group loading, caching
- Each class-loader maintains its own *namespace*
- Errors include: ClassFormatError, UnsupportedClassVersionError, ClassCircularityError, NoClassDefFoundError

Class Loaders

- System classes are automatically loaded by the bootstrap class loader
- To see which: `java -verbose:class Test.java`
- Arrays are created by the VM, not by a class loader
- A class is *unloaded* when its class loader becomes unreachable (bootstrap class loader is never unreachable)

Class Linking - 1. Verification

- Extensive checks that the .class file is valid
- This is a **vital** part of the JVM security model
- Needed because of possibility of:
  - buggy compiler, or no compiler at all
  - malicious intent
  - (class) version skew
- Checks are independent of compiler and language

More later...

Class Linking - 2. Preparation

- Create *static* fields for a class
- Set these fields to the standard default values (N.B. not explicit initialisers)
- Construct *method tables* for a class
- ... and anything else that might improve efficiency

Class Linking - 3. Resolution

- Most classes refer to methods/fields from other classes
- Resolution translates these *names* into explict *references*
- Also checks for field/method existence and whether access is allowed

Class Initialisation

Happens **once** just before first instance creation, or first use of static variable.

- Initialise the superclass first!
- Execute (class) static initialiser code
- Execute explicit initialisers for static variables
- May not need to happen for use of *final* static variable

Completed before anything else sees this class

Instance Creation/Finalisation

- Instances are created using `new`, or `newInstance()` from class `Class`
- Instances of `String` may be created (implicitly) for String literals
- Process:
  1. Allocate space for all the instance variables (including the inherited ones),
  2. Initialise them with the default values
  3. Call the appropriate constructor (do parent's first)
- `finalize()` called just before garbage collector takes the object (so timing is unpredictable)

JVM Architecture

The internal runtime structure of the JVM consists of:

- One: (i.e. shared by all threads)
  - method area
  - heap
- For each thread, a:
  - program counter (pointing into the method area)
  - Java stack
  - native method stack (system dependent)

The Method Area

- Contains one entry for each class
- Lists all details relating to that class
- Includes the *constant pool*
- Contains the *code* for the methods
- May grow/shrink as classes are loaded/unloaded

Shared by all threads.

The Heap

- One entry for each object
- Increases with each instance creation
- Decreases with garbage collection (mechanism not specified)
- Object information: instance field values, pointer to class, locking info, virtual method table(?)

Shared by all threads.

Java Stack

- JVM pushes and pops **frames** onto this stack
- Each frame corresponds to the invocation of a method
- Call a method $\rightarrow$ push its frame onto the stack
- Return from a method $\rightarrow$ pop its frame
- Frame holds parameter values, local variables, intermediate values etc.