Advanced Programming in Java
CM2: the Java Virtual Machine (JVM)

Arthur Bit-Monnot

INSA 4IR

Java Virtual Machine: Motivation

```
// C: 0.87 ns / element
int max(int* ints) {
    int m = 0;
    for (long i = 0; i<N; i++) {
        int curr = ints[i];
        m = m > curr ? m : curr;
    }
    return m;
}
```

```
# python: 240.3 ns / element
m = 0
for i in ints:
    m = max(m, i)
```

Java Virtual Machine: Motivation

```c
// C: 0.87 ns / element
int max(int* ints) {
    int m = 0;
    for (long i = 0; i<N; i++) {
        int curr = ints[i];
        m = m > curr ? m : curr;
    }
    return m;
}
```

```python
# python: 240.3 ns / element
m = 0
for i in ints:
    m = max(m, i)
```

```java
// java (ints) 0.97 ns / element
static int maxInt(int[] ints) {
    int max = 0;
    for (int i=0; i< ints.length; i++) {
        max = Math.max(max, ints[i]);
    }
    return max;
}
```

Java Virtual Machine: Motivation

```c
// C: 0.87 ns / element
int max(int* ints) {
    int m = 0;
    for (long i = 0; i<N; i++) {
        int curr = ints[i];
        m = m > curr ? m : curr;
    }
    return m;
}
```

```python
# python: 240.3 ns / element
m = 0
for i in ints:
    m = max(m, i)
```

```java
// java (ints) 0.97 ns / element
static int maxInt(int[] ints) {
    int max = 0;
    for (int i=0; i< ints.length; i++) {
        max = Math.max(max, ints[i]);
    }
    return max;
}
// java (Integer) 2.68 ns / element
static Integer maxInteger(Integer[] ints) {
    int max = 0;
    for (int i=0; i< ints.length; i++) {
        max =  Math.max(max, (int) ints[i]);
    }
    return max;
}
```

The JVM is what explains 90% of the difference between Java and Python.

Compiling a java program

javac: the Java Compiler

```
javac json.java
```

- produces one `.class` file for each java class in the source file
  - `Json.class`
  - `Json$Point.class`
  - `Json$Vector.class`

Note: sometime java projects are compiled as a `.jar` file:

- simply a **zip** file containing `.class` files !

Class files & Java Bytecode

Each class file contains the description of the class in a **binary format** that is easily consumed by machines

- the class metadata
- implemented classes and interfaces
- all class attributes
- all provided methods

The *java source code* in the methods in translated to **java bytecode**.

java classfile (human readable view)

```
// class version 64.0 (64)
// access flags 0x20
class test$MyInt {

  // compiled from: test.java
  NESTHOST test
  // access flags 0xA
  private static INNERCLASS test$MyInt test MyInt

  // access flags 0x11
  public final I n

  // access flags 0x2
  private <init>(I)V
   L0
```

The Java Bytecode

- Instruction set of the JVM
- Build for efficient machine analysis
- Closely matches the Java language

All languages running on the JVM are compiled to java bytecode

- Java, Scala, Kotlin, Clojure, . . .

java bytecode (human readable view)

```
// access flags 0x1
public toString()Ljava/lang/String;
 L0
  LINENUMBER 14 L0
  ALOAD 0
  GETFIELD test$MyInt.n : I
  INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
  ASTORE 1
 L1
  LINENUMBER 15 L1
  ALOAD 1
  INVOKEVIRTUAL java/lang/Integer.toString ()Ljava/lang/String;
  ARETURN
 L2
  LOCALVARIABLE this Ltest$MyInt; L0 L2 0
  LOCALVARIABLE nBoxed Ljava/lang/Integer; L1 L2 1
  MAXSTACK = 1
  MAXLOCALS = 2
```

The java Runtime

```
java Json --class-path .
> {x: 10,y: 34,}
> [{x: 10,y: 23,}, {x: 1,y: 2,}, ]
> {start: {x: 10,y: 23,},end: {x: 1,y: 2,},}
```

> *The java command starts a Java application. It does this by starting the Java Virtual Machine (JVM), loading the specified class, and calling that class's 'main()" method.*

- `Json` : name of the class to execute
- `--class-path .` : path to the compiled class files

Section 1

Executing Java programs: Interpreter & JIT Compiler

Class loading

Upon starting, the JVM will start by *loading the necessary classes**

- starting from the **main class** (specified on the command line)
- recursively loading all classes it encounters

If a class is not found in the classpath, it throws `ClassNotFoundException` and exits.

- indicates that the classpath is wrong

After this phase, all the code necessary for your program to run is available in the memory (RAM, in the JVM process).

The interpreter

Problem: java bytecode != CPU instruction set

- not immediately executable

The interpreter

Problem: java bytecode != CPU instruction set

- not immediately executable

The java bytecode is **interpreted**:

- a program reads the bytecode the bytecode line by line and simulates its executation

dummy bytecode interpreter

```
for ever
  inst := load_next_instruction
  if inst == "i_load"
    value := read_from_ram(...)
    push(value)
  elif inst == "i_add"
    v1 := pop()
    v2 := pop()
    result := v1 + v2
    push(result)
  ...
```

Beyond Interpreters: the JIT Compiler

Problem of interpreters: they are *slooooow*

- easily 10x slower than optimized machine code

Just In Time (JIT) compilation:

- if a method is called more than a given threshold, compile to efficient machine code (x86, ARM, ...)

Advantages of JIT compilers

- by compiling at the last possible moment, the JVM knows **a lot** about current program
- precise architecture of the current machine
- statistics on the method currently executed
  - number of invocation of methods
  - actual class of the parameters
  - branches taken

The interpreter plays a crucial role in gathering the statistics for a more efficient compilation! (profiling)

Advantages of JIT: Optimistic Optimization & De-Optimization

Very often, profiling suggests very interesting optimizations

> *"So far, the `toJson` method has ALWAYS been called with the `Point` class as parameter"*

Optimistic Optimization:

- compile as if your assumption was always true, (`toJson` will always be called with `Point`)
  - enables many optimizations

Advantages of JIT: Optimistic Optimization & De-Optimization

Very often, profiling suggests very interesting optimizations

> *"So far, the `toJson` method has ALWAYS been called with the `Point` class as parameter"*

Optimistic Optimization:

- compile as if your assumption was always true, (`toJson` will always be called with `Point`)
  - enables many optimizations
- at the beginning of the generated code, add a check that your assumption hold (a guard)
- if the check fails, throw away the compiled code and go back to the interpreter (de-optimization)
- you can later recompile with the additional information (e.g. `toJson` is called with `Point` 97% of the time)

In other languages[1]

Language	Pre-compiled	Interpreted	Interpreted + JIT Compiled
Java			X
C / C++	X		
Python		X	
Javascript			X
Rust	X		
Go	X		
Bash		X	

[1] In main implementation. For instance, PyPy provides a JIT for python but is incompatible with a large part of the ecosystem.

Section 2

Memory Management: Garbage Collection

Memory Management: Garbage Collection

```
void allocatingMethod() {
  var n = new Integer(11);
  doSomething(n);
}
```

What happened to the memory allocated for `n` ?

Memory Management not always trivial

```java
// pushed to a data structure
// that outlives the method
void allocatingMethod(
    List<Integer> collection)
{
  var n = new Integer(11);
  collection.add(n);
}

// sent to another thread
void allocatingMethod() {
  var n = new Integer(11);
  Thread t = new MyThread(n)
  t.start();
}
```

```java
// nested in a more complex datastructure
void allocatingMethod() {
  var x = new Integer(11);
  var y = new Integer(14);
  var point = new Point(x, y);
  ...
}
```

Memory Management Approaches: Manual Management

Let the programmer decide when to deallocate

```
int * onHeapInt = malloc(sizeof(int));
...
free(onHeapInt);
```

Taken historically by low-level languages (C)

- most powerful and potentially efficient

Endless source of bugs and security issues

- memory leaks
- use after free
- double free

Memory Management Approaches: Reference Counting

Principle: in every object header, add a counter that keeps track of the number of references to this object

- each time a reference is copied: increment the counter
- each time a reference goes out of scope: decrement the counter
- if the counter reaches 0, deallocate the object

Memory Management Approaches: Reference Counting

Principle: in every object header, add a counter that keeps track of the number of references to this object

- each time a reference is copied: increment the counter
- each time a reference goes out of scope: decrement the counter
- if the counter reaches 0, deallocate the object

Problems:

- each time a reference is copied/deleted an **atomic increment** must be made to the object counter! (costly)
- if there is a cycle of references, the memory will not be freed (memory leak)

Adopters: Python, Swift, C++ (with smart pointers)

Memory Management Approaches: Tracing Garbage Collector

(Basic) Approach: at regular intervals

- stop the program
- **mark**: from the references on the stack, recursively follow all references and *mark* all objects you encounter
- **sweep**: go through the entire program's memory and remove all objects that are not marked
- resume the program

General principle: **tracing garbage collection (GC)** (here with the **mark-and-sweep** algorithm).

- introduced by Lisp in 1959 (xkcd/297)
- adopters: Java, JavaScript, Go

Memory Management Approaches: Tracing Garbage Collector

GC much more evolved nowadays:

- concurrent: no stopping for marking / sweeping
- compacting: reorganize allocated memory on sweep (require stopping)
- generational: differentiated handling of short-lived / long-lived objects

Nowadays:

- robust and correct
- very efficient: throughput comparable with manual memory management[2]
  - gained by compaction: make memory cache-firendly and allocations trivial
- downsides:
  - cpu hoverhead (low)
  - memory overhead (significant)
  - short program pauses: impacts worse-case latency

[2] of course, manual management offers more opportuinities for optimization

Memory Management: Languages Status

Language	Manual	Ref-counted	Tracing GC
Java			X
C / C++	X		
Python		X	X
Javascript			X
Go			X
OCaml			X
Swift		X	
Rust[3]			

[3] compiler enforces and tracks a single *owner* for each object. Correct and with no overhead !!!

Section 3

Summary

Summary

The JVM has several mechanisms to

- Interpreter + JIT compiler
  - reach peak performance
- Garbage Collection
  - Analysis references to objects to decide when to deallocate an object
  - optimize memory allocations

Downsides

- class loading impacts the startup time of the JVM
  - can be a few seconds for programs with huge dependencies
- it takes time to reach peak performance
  - requires runtime analysis + JIT compilation
- GC requires to stop the program regularly
  - may induce latencies
- important memory consumption overhead (GC + class loading)

Java is not suitable for every task

- heavily used in server environments for long-running tasks
- unsuitable in applications where:
  - memory is scarce (embedded)
  - require fine-grained memory control (OS/drivers)
  - are very short-lived (command line tools)