

What does the JVM do with my code?

Manas Thakur

PACE Lab, IIT Madras



Language Translator

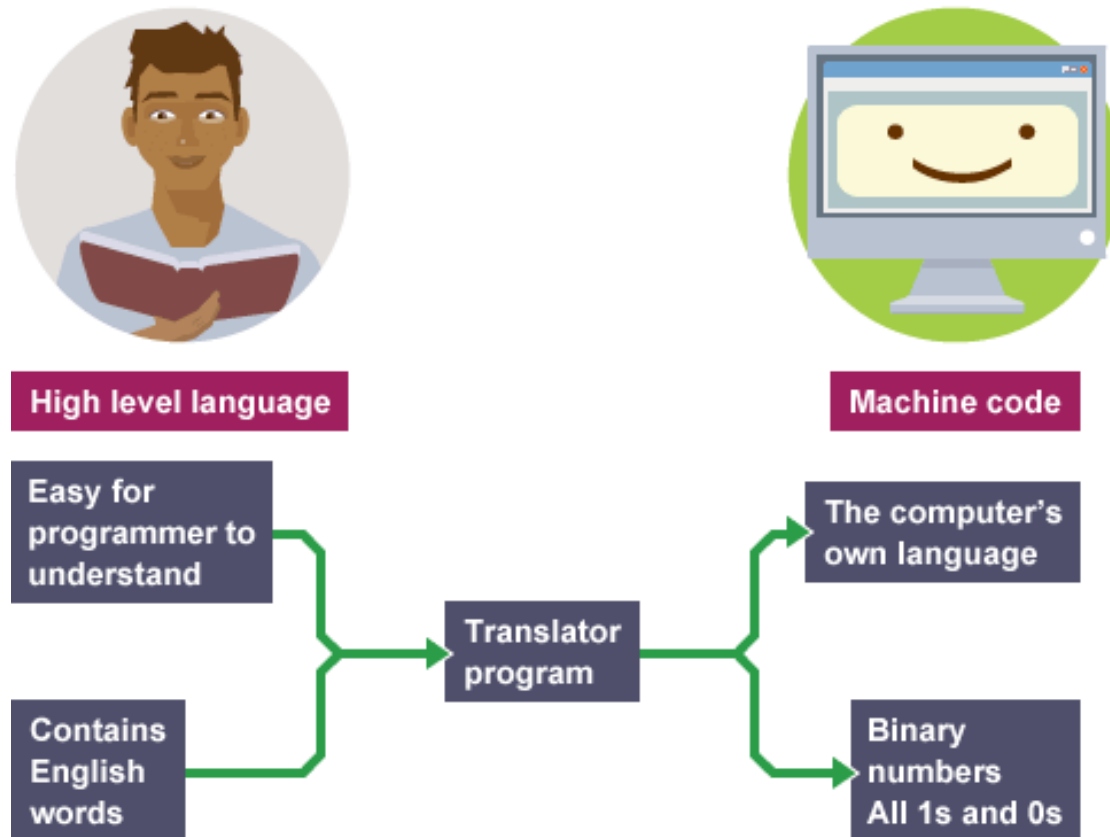


Image source: <http://www.bbc.co.uk/education/guides/zgmpr82/revision>



Compiler vs Interpreter

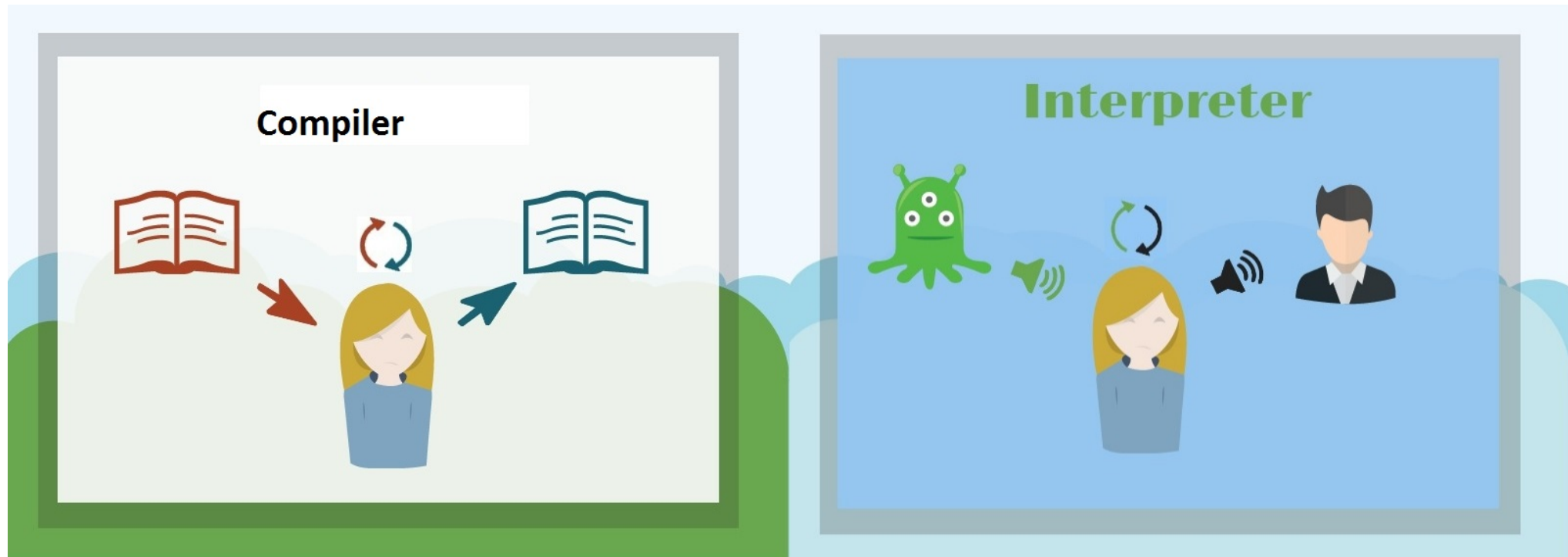
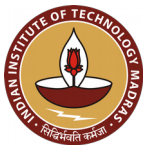


Image source: <https://stackoverflow.com/a/31551282>



Compiler vs Interpreter

A COMPILER

Input ... takes an entire program as its input.

Output ... generates intermediate object code.

Speed ... executes faster.

Memory ... requires more memory in order to create object code.

Workload ... doesn't need to compile every single time, just once.

Errors ... displays errors once the entire program is checked.

AN INTERPRETER

... takes a single line of code, or instruction, as its input.

... does not generate any intermediate object code.

... executes slower.

... requires less memory (doesn't create object code).

... has to convert high-level languages to low-level programs at execution.

... displays errors when each instruction is run.

Image source: <https://www.upwork.com>



Outline

- Basics
- The Java way
- HotSpot under the hood
- Playing around

HONEST JON

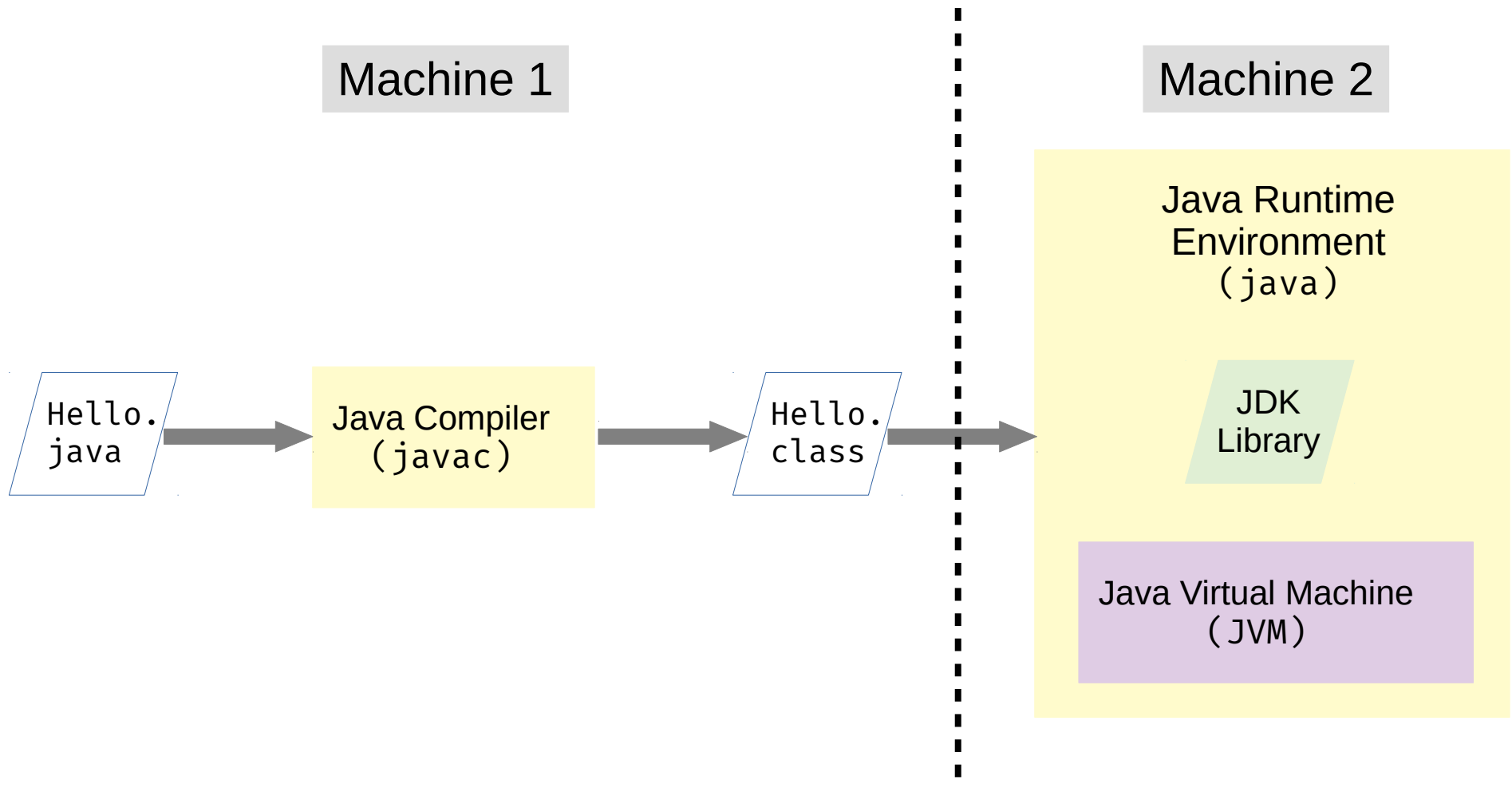
by Jon Clark



Ways to begin a talk: The Overdone Overview



The Java Compilation+Execution Model



A Bit of Bytecode

```
int a = 10;  
int b = 20;  
int c = a + b;
```

```
0: bipush      10  
2: istore_1  
3: bipush      20  
5: istore_2  
6: iload_1  
7: iload_2  
8: iadd  
9: istore_3  
10: return
```

Bytecode indices

`javap -c class_name`



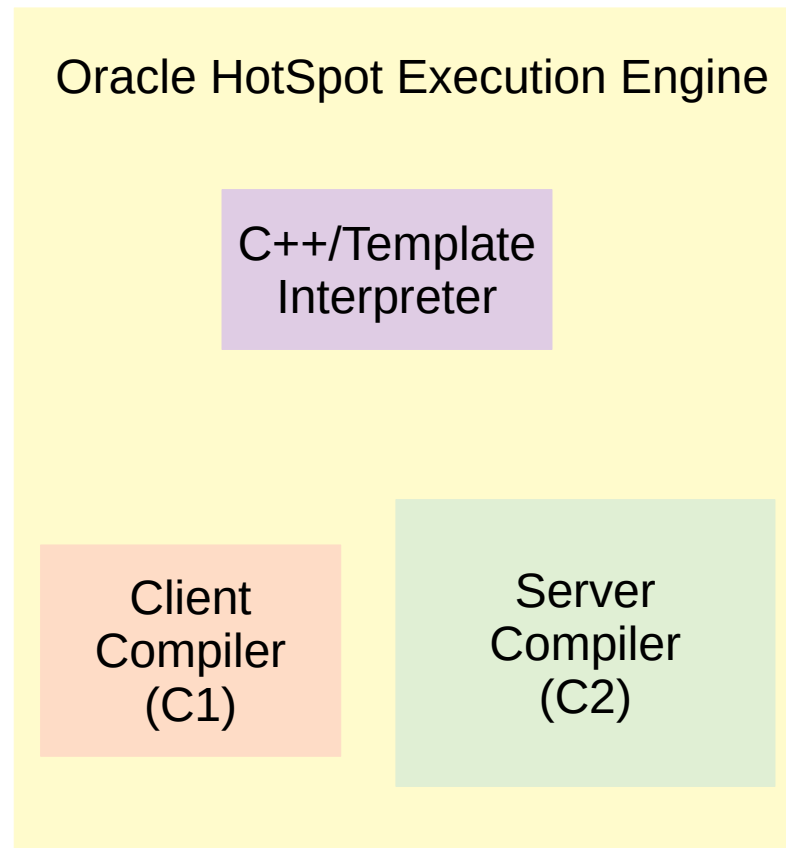
What does the JVM do with my code?

- Basics
- The Java way
- HotSpot under the hood
- Playing around



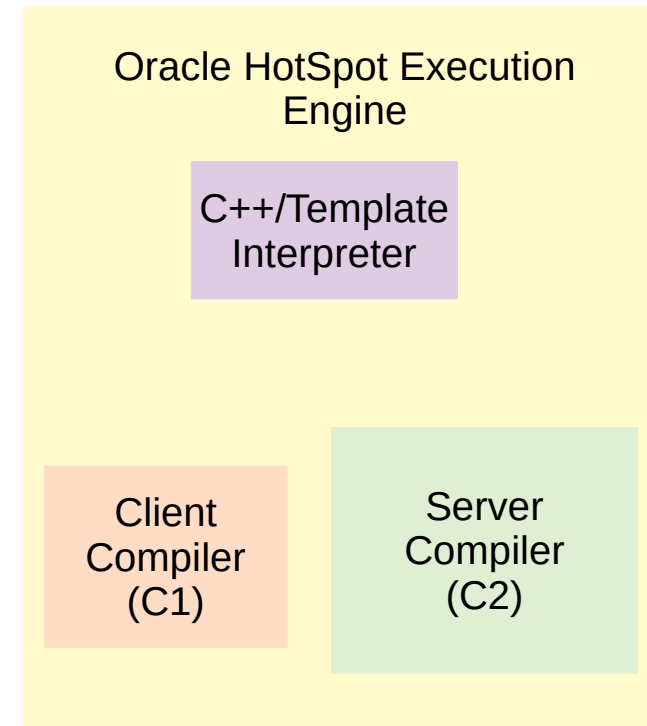
Is Java Bytecode interpreted or compiled?

Java Bytecode is interpreted as well as compiled!!



The “HotSpot” JVM

- HotSpot uses *tiered* compilation with profiling
 - Starts off with interpreter
 - Hot spots get compiled as they get executed
 - Method entry-points changed dynamically
 - Loops replaced *on-the-stack*
- Interpreters:
 - C++ interpreter (deprecated)
 - Template interpreter
- Just-In-Time (JIT) Compilers:
 - C1 (aka *client*)
 - C2 (aka *server*)



The C++ Interpreter

- Simple switch-case

```
switch (bytecode) {  
    case nop          : break;  
    case aconst_null: push(null); break;  
    case iconst_1     : push(1); break;  
    ...  
}
```

- Disadvantage: **Slow**
 - Too many comparisons
 - No idea where to go for the next bytecode



The C1 Compiler

- Targets fast compilation
- Still performs several optimizations:
 - Method inlining
 - Dead code/path elimination
 - Heuristics for optimizing call sites
 - Constant folding
 - Peephole optimizations
 - Linear-scan register allocation, etc.
- Threshold: 1000 to 2000



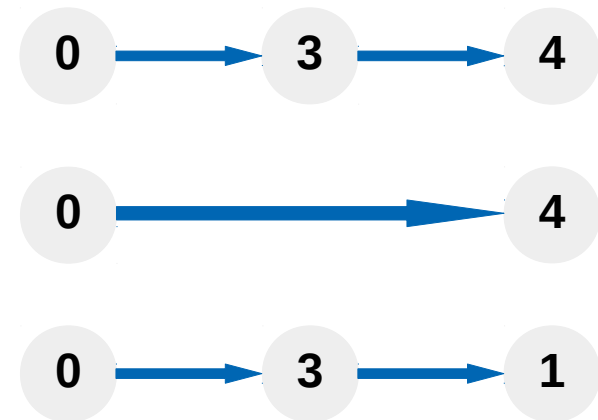
The C2 Compiler

- Targets more-and-more optimization
- Performs expensive optimizations (*apart from the ones performed by C1*):
 - Escape analysis
 - Null-check elimination
 - Loop unrolling/unswitching
 - Branch prediction
 - Graph-coloring based register allocation, etc.
- Threshold: 10000 to 15000



Compilation Levels

- 0 – Interpreter
- 1 – Pure C1
- 2 – C1 with invocation and backedge counting
- 3 – C1 with full profiling
- 4 – C2 (full optimization)



Deoptimization

- Optimistic optimizations:
 - Branch prediction
 - Implicit null checks
 - Morphism
- When an assumption fails, the compiled method may be invalidated, and the execution falls back to the interpreter
- Consistency maintained using *safepoints*
- Method states: in use, not entrant, zombie, unloaded

Deoptimization is costly; happens lesser the better



HotSpot in Action



When Theory becomes Practice

- Basics
- The Java way
- HotSpot under the hood
- Playing around



"It was here when Harris decided to 'tweak' things a bit..."

Some Important Flags: Compilation

- Compilation details: `-XX:+PrintCompilation`
- Dump assembly: `-XX:+PrintInterpreter`
- Interpreter-only mode: `-Xint`
- Compiler-only mode: `-Xcomp`
- Disable levels 1, 2, and 3: `-XX:-TieredCompilation`
- Stop compilation at level n: `-XX:TieredStopAtLevel=n`



Some key learnings

- Java programs are not inherently slow.
- Compiler analyses/optimizations tremendously affect the program performance.
- Java programs are interpreted *as well as* compiled.
- Trust the JVM, and help it.
- Keep experimenting.



Pointers for the enthusiast

- <https://www.cubrid.org/blog/understanding-jvm-internals>
- <https://www.artima.com/insidejvm/ed2/jvmP.html>
- <https://declara.com/content/3gBB6Jge>
- <https://www.infoq.com/presentations/hotspot-memory-data-structures>
- <http://www.progdoc.de/papers/Jax2012/jax2012.html>
- <https://www.ibm.com/developerworks/library/j-jtp12214/index.html>



Stay Hungry, Stay Foolish, Stay Connected



github.com/manasthakur
gist.github.com/manasthakur

www.cse.iitm.ac.in/~manas
manasthakur.github.io



manasthakur17@gmail.com



manasthakur.wordpress.com

linkedin.com/in/manasthakur



www.cse.iitm.ac.in/~manas/docs/cs6843-hotspot.pdf

