What does the JVM do with my code?

Manas Thakur
PACE Lab, IIT Madras

Content Credits

- *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*

Language Translator

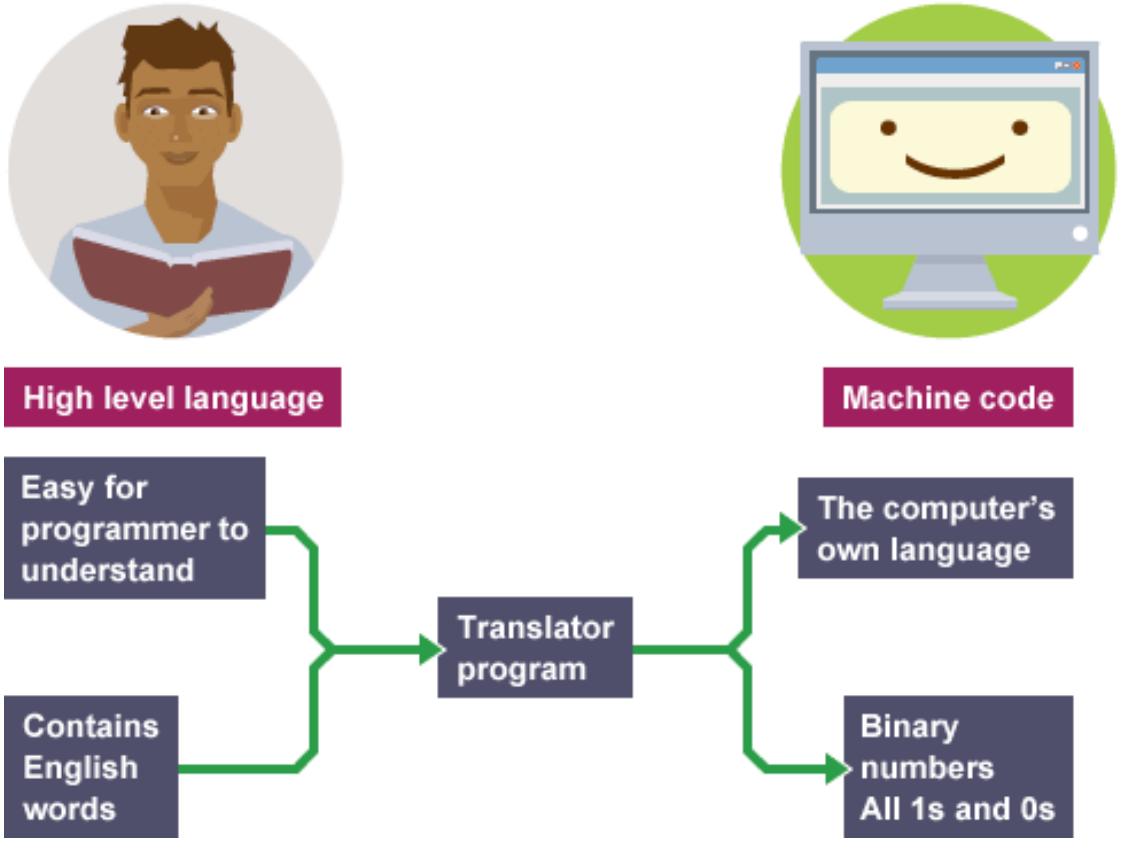


Image source: http://www.bbc.co.uk/education/guides/zgmpr82/revision

Compiler vs Interpreter

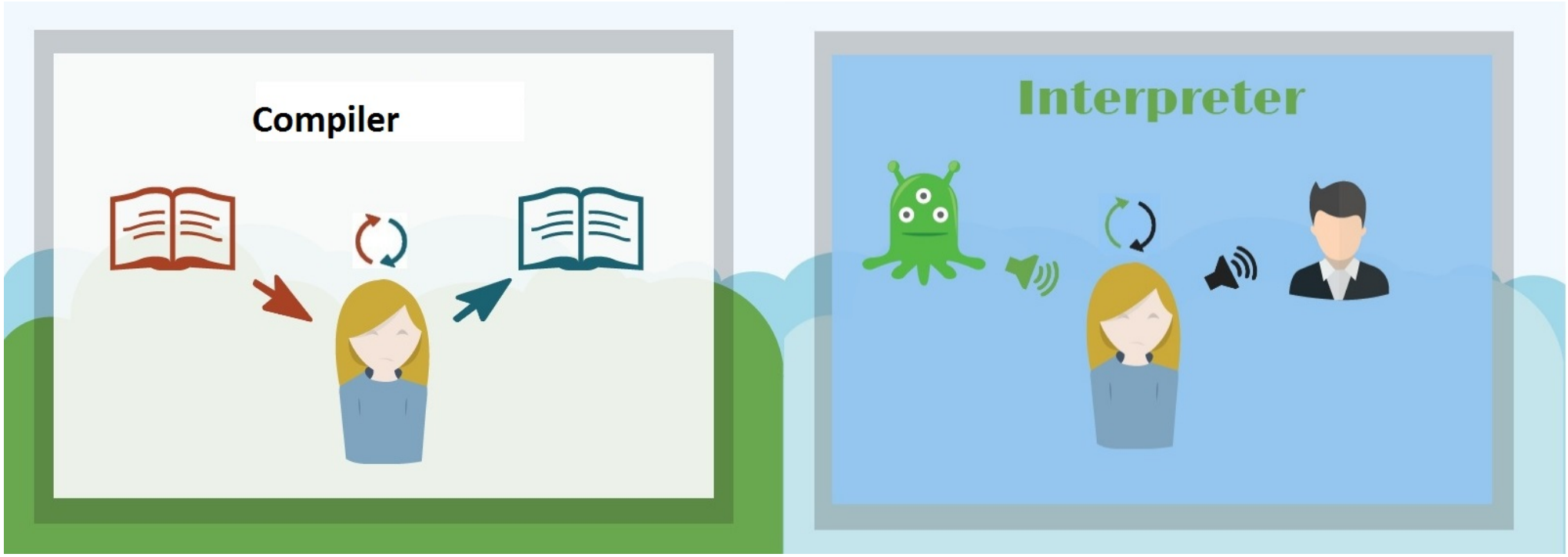


Image source: https://stackoverflow.com/questions/2377273/how-does-an-interpreter-compiler-work

Compiler vs Interpreter

	A COMPILER	AN INTERPRETER
Input	... takes an entire program as its input.	... takes a single line of code, or instruction, as its input.
Output	... generates intermediate object code.	... does not generate any intermediate object code.
Speed	... executes faster.	... executes slower.
Memory	... requires more memory in order to create object code.	... requires less memory (doesn't create object code).
Workload	... doesn't need to compile every single time, just once.	... has to convert high-level languages to low-level programs at execution.
Errors	... displays errors once the entire program is checked.	... displays errors when each instruction is run.

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

Outline

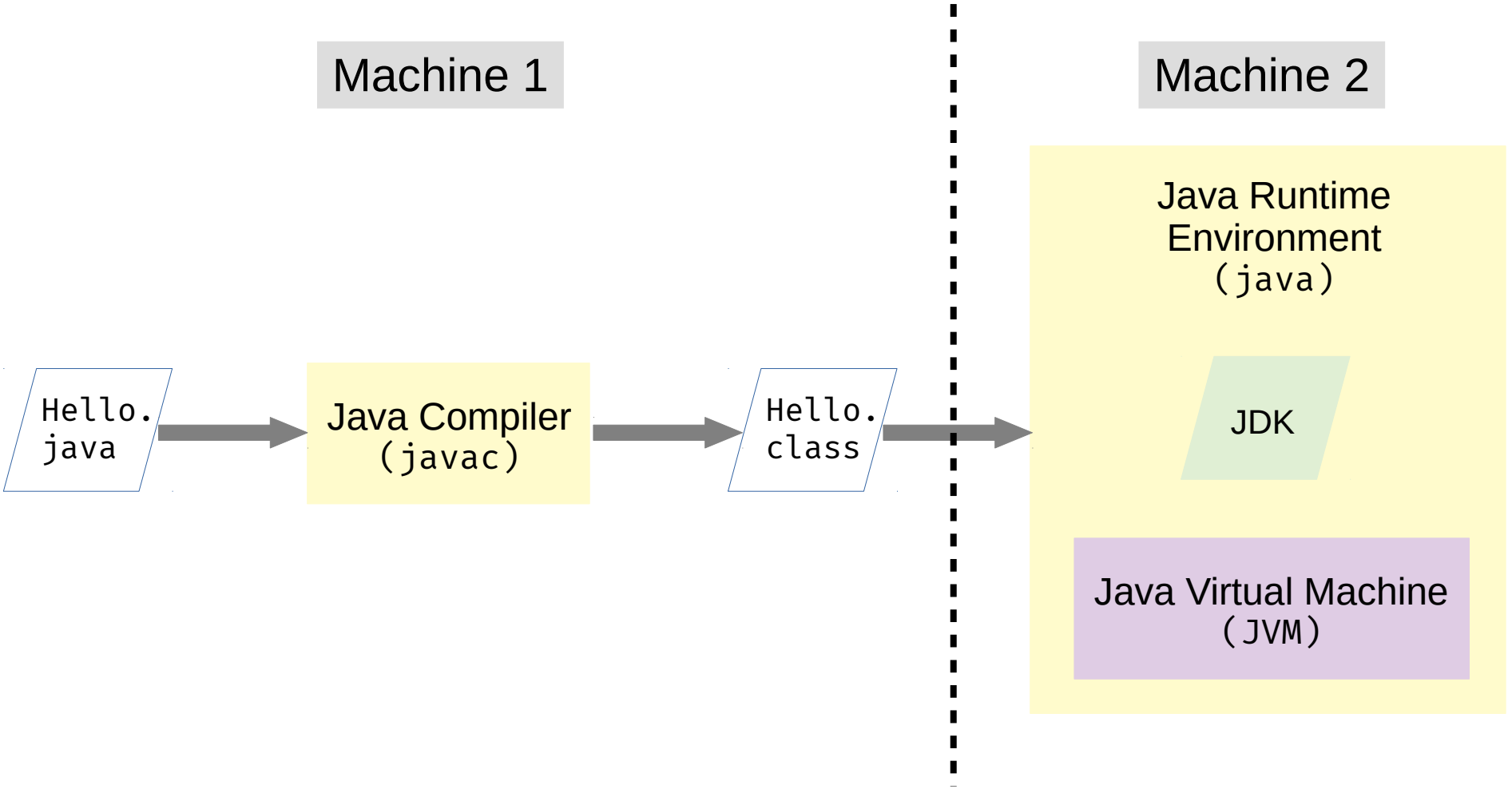
- Basics
- The Java way
- JVM architecture
- HotSpot under the hood
- Playing around
- Being a better Java programmer

Ways to begin a talk: The Overdone Overview

Java SE Version History (Green: Major; Blue: Minor)

Dates from http://en.wikipedia.org/wiki/Java_version_history and http://oracle.com.edgesuite.net/timeline/java/[5]

The Java Compilation+Execution Model



Quiz-1: Which implementation is better?

```java
Tokens obj = new Tokens();

String str = "";
str += obj.getToken(0);
str += obj.getToken(1);
str += obj.getToken(2);
str += obj.getToken(3);
str += "?";

System.out.println(str);
```

```java
Tokens obj = new Tokens();

StringBuilder str = new StringBuilder();
str.append(obj.getToken(0));
str.append(obj.getToken(1));
str.append(obj.getToken(2));
str.append(obj.getToken(3));
str.append("?");

System.out.println(str);
```

A Bit of Bytecode

- Generated by the static Java compiler
- Mid-level IR
- Machine independent
- Follows a stack model
- Format:

```
<opcode> <operands>
```

- Opcode is one *Byte* (8 bits)
  - 256 (2^8) in number

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

A Bit of Bytecode

```
A obj = new A(10);
int objA = obj.getA();
System.out.println(objA);
```

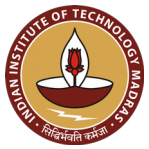
```
 0: new           #2                  // class A
 3: dup
 4: bipush        10
 6: invokespecial #3                  // Method A."<init>":(I)V
 9: astore_1
10: aload_1
11: invokevirtual #4                  // Method A.getA:()I
14: istore_2
15: getstatic     #5                  // Field java/lang/System.out:Ljava/io/PrintStream;
18: iload_2
19: invokevirtual #6                  // Method java/io/PrintStream.println:(I)V
22: return
```

Method invocations

Type expression

Type Expressions

Java Bytecode	Type	Description
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L<classname>	reference	an instance of class <classname>
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension



Method invocations

- invokespecial
- invokestatic
- invokeinterface
- invokevirtual
- invokedynamic

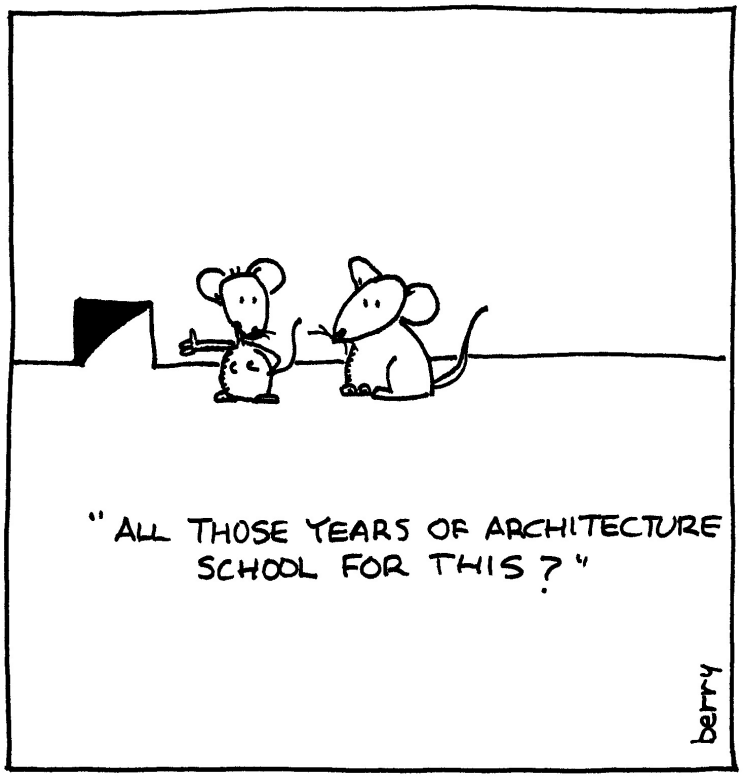
Class file format

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];}
```

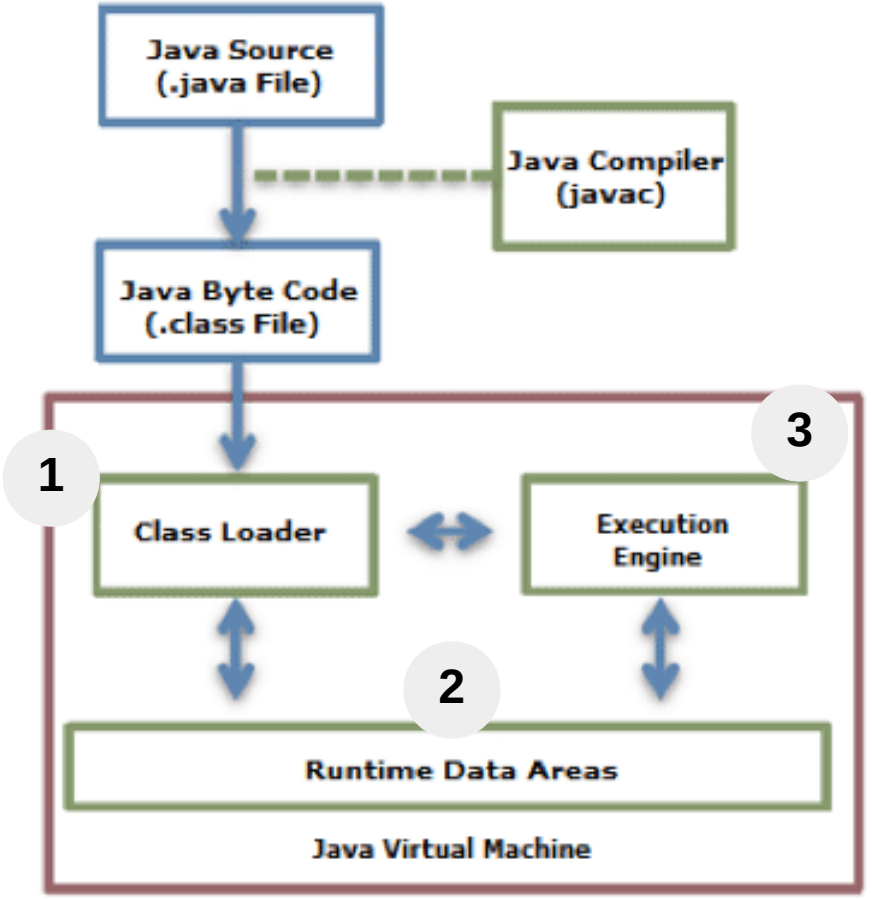
The Java Class File Disassembler (`javap`)

All Problems are Solved by Good Design

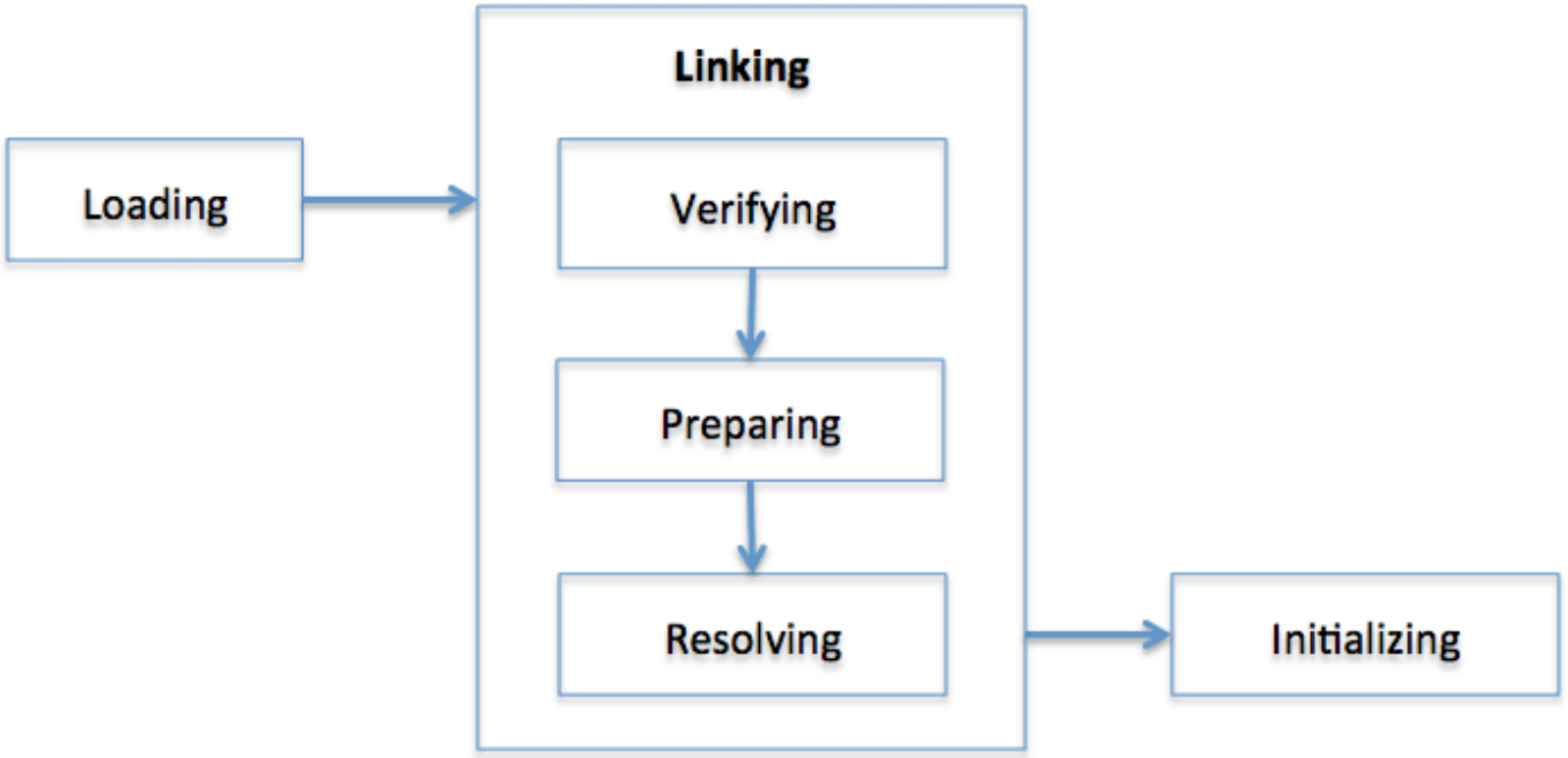
- The Java way
- The Java (static) compiler
- JVM architecture
- HotSpot under the hood
- Playing around
- Being a better Java programmer



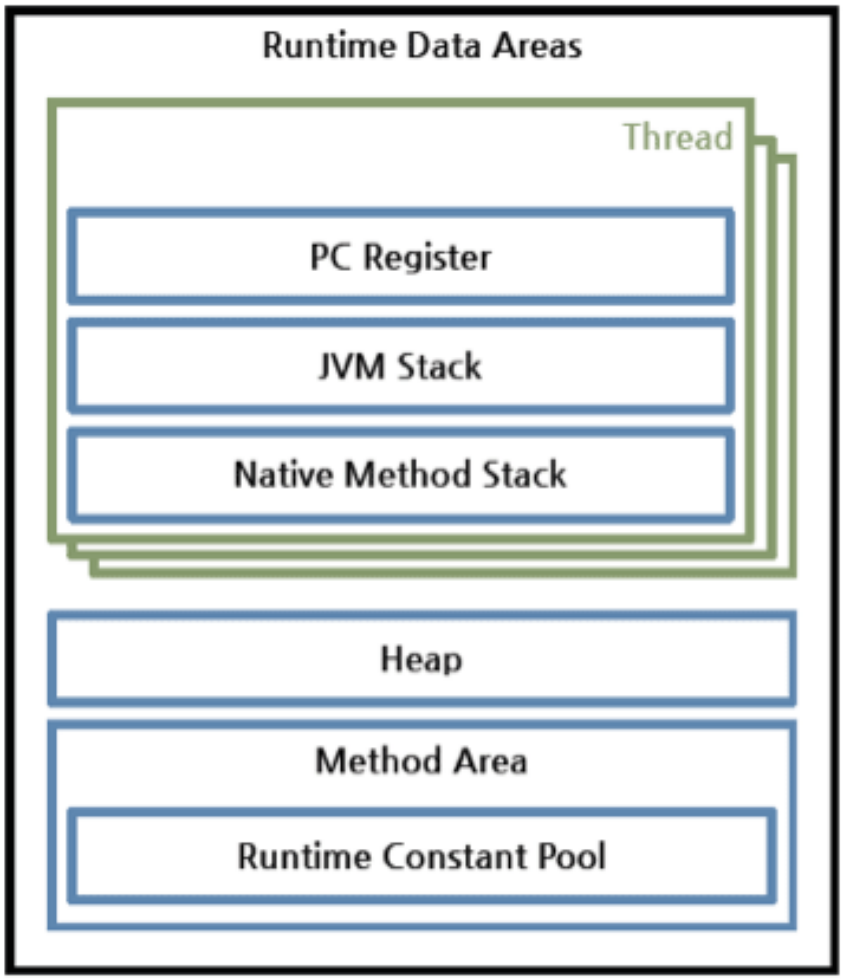
Diving slightly deeper



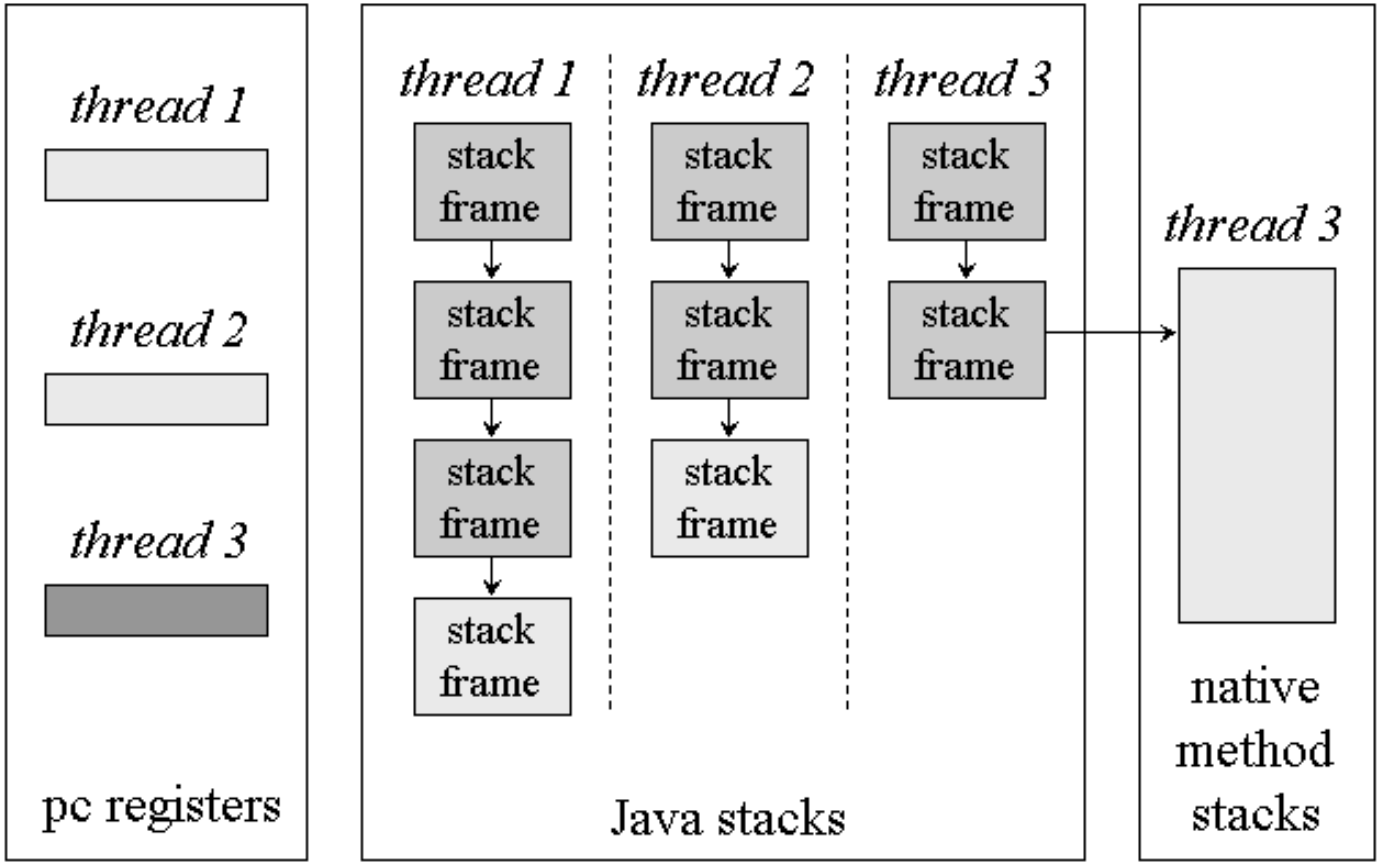
The Class Loader



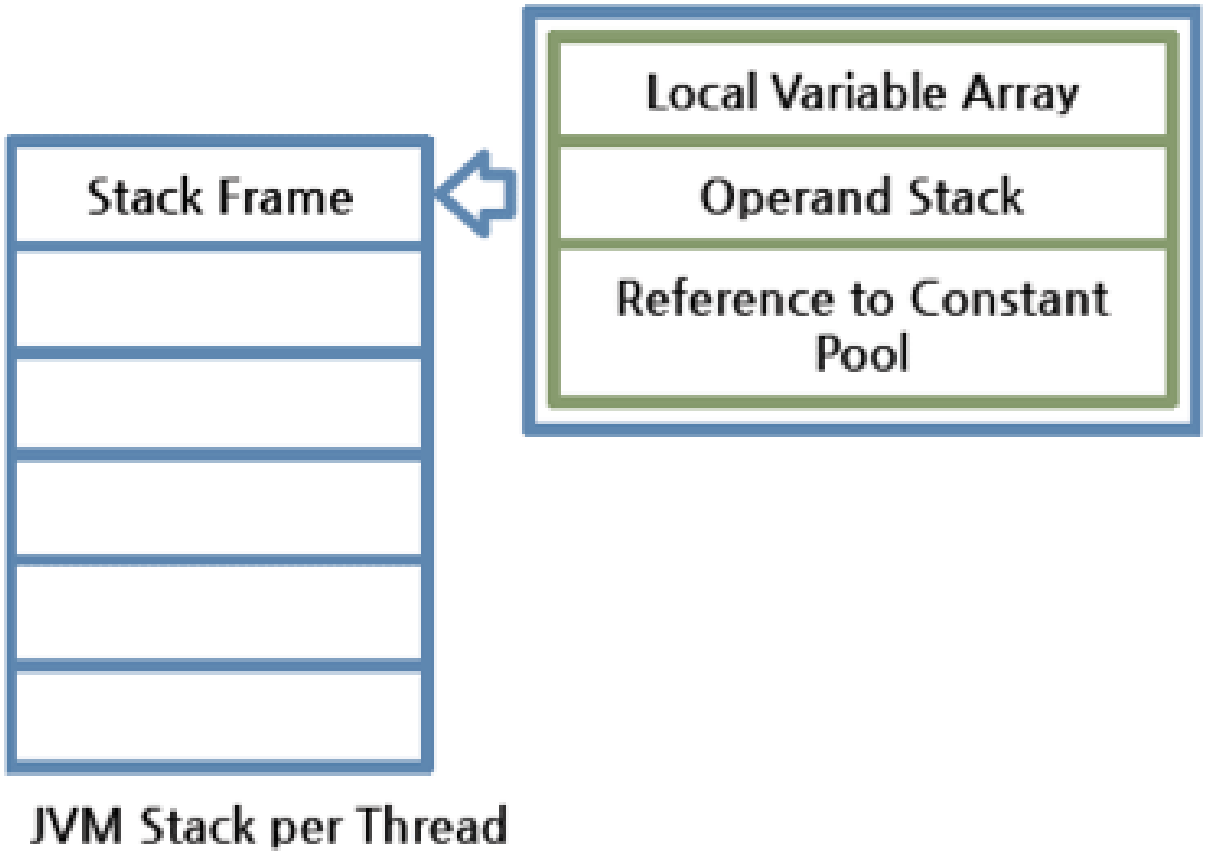
Runtime Data Areas



Thread-local runtime data



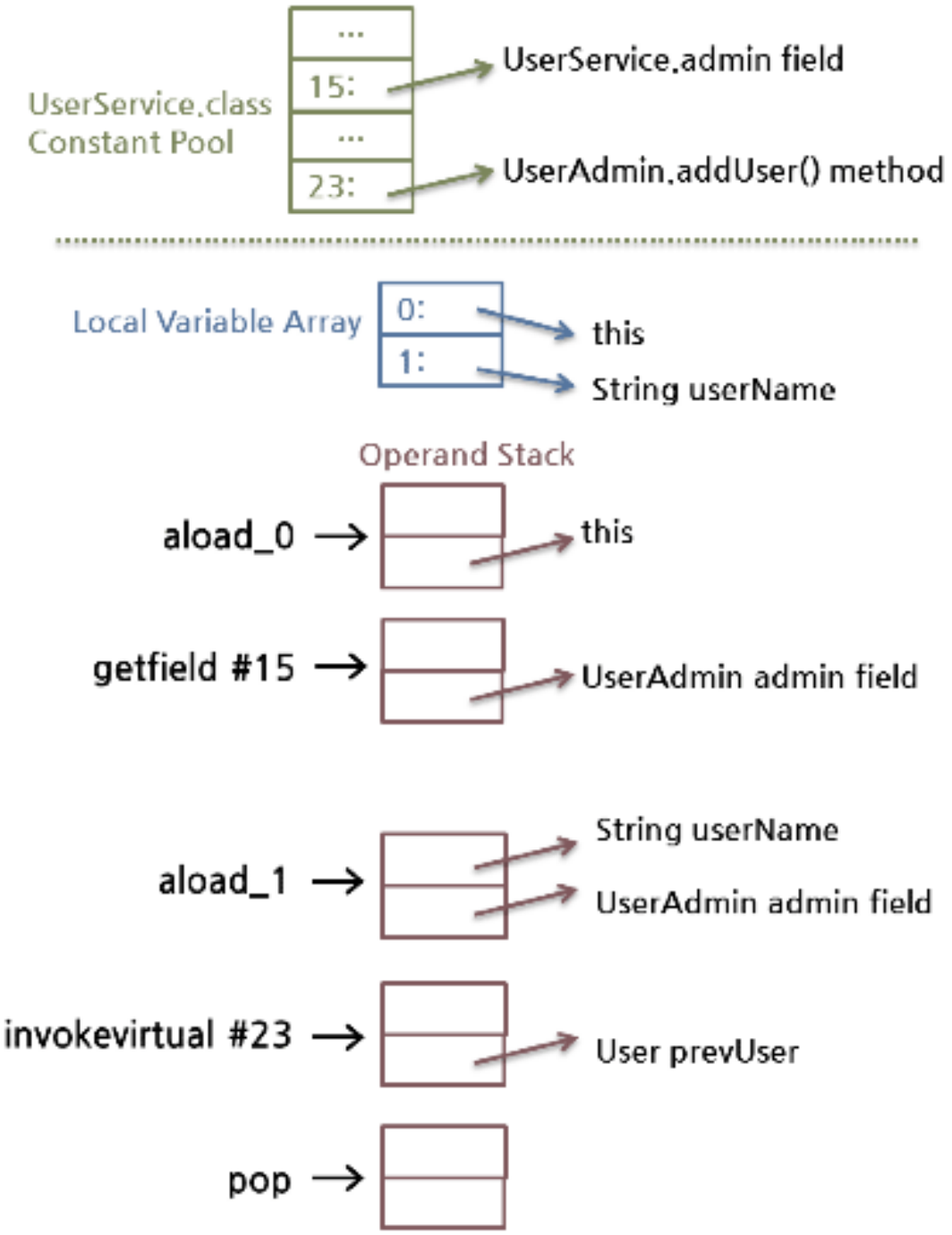
Stack Frames



Stack Frames

```
// class 'UserService'
public void add(String userName) {
    admin.addUser(userName);
}
```

```
0:    aload_0
1:    getfield        #15
4:    aload_1
5:    invokevirtual   #23
8:    pop
9:    return
```



Before we leave: Which implementation is better?

```java
Tokens obj = new Tokens();

String str = "";
str += obj.getToken(0);
str += obj.getToken(1);
str += obj.getToken(2);
str += obj.getToken(3);
str += "?";

System.out.println(str);
```

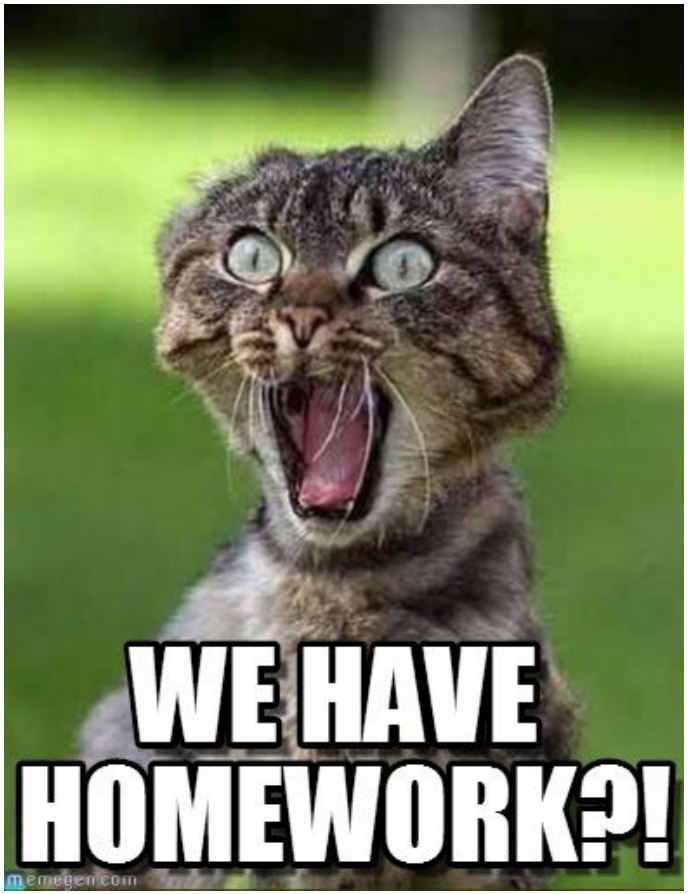
```java
Tokens obj = new Tokens();

StringBuilder str = new StringBuilder();
str.append(obj.getToken(0));
str.append(obj.getToken(1));
str.append(obj.getToken(2));
str.append(obj.getToken(3));
str.append("?");

System.out.println(str);
```

Homework: Build OpenJDK on a Linux machine

- Install dependencies:
  - `mercurial, alsa, freetype, cups, xrender, g++, java`
- Clone OpenJDK sources:
  - `hg clone http://hg.openjdk.java.net/jdk8/jdk8 YourOpenJDK`
  - `cd YourOpenJDK`
  - `bash ./get_source.sh`
- Configure and build (will take about half an hour):
  - `bash ./configure`
  - `make images`
- Test the new Java binaries:
  - `YourOpenJDK/build/linux-*-release/images/j2sdk-image/bin/javac -version`
  - `YourOpenJDK/build/linux-*-release/images/j2sdk-image/bin/java -version`



(For details, visit: `hg.openjdk.java.net/build-infra/jdk8/raw-file/tip/README-builds.html`)

Homework-2: Abra-ca-dabra

- Install a hex-editor
  (say GNOME Hex or Bless).
- Open a .class file with the hex editor.
- Find out the magic number
  (recall? first 4 bytes).
- Check it for some other .class files.
- If you find something interesting,
  Google it and let me know.

Homework slides available at: ***manasthakur.github.io/docs/hw.pdf***

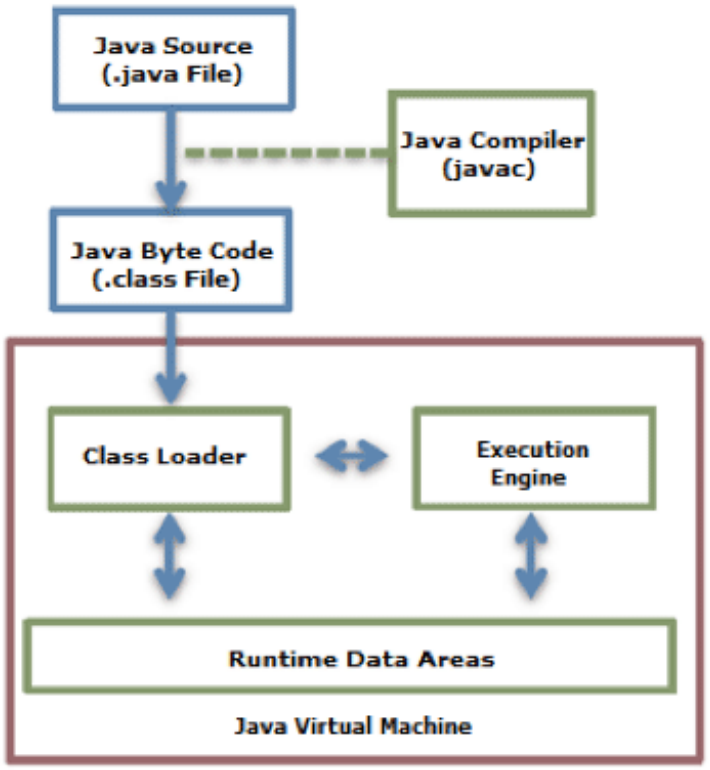
What does the JVM do with my code?

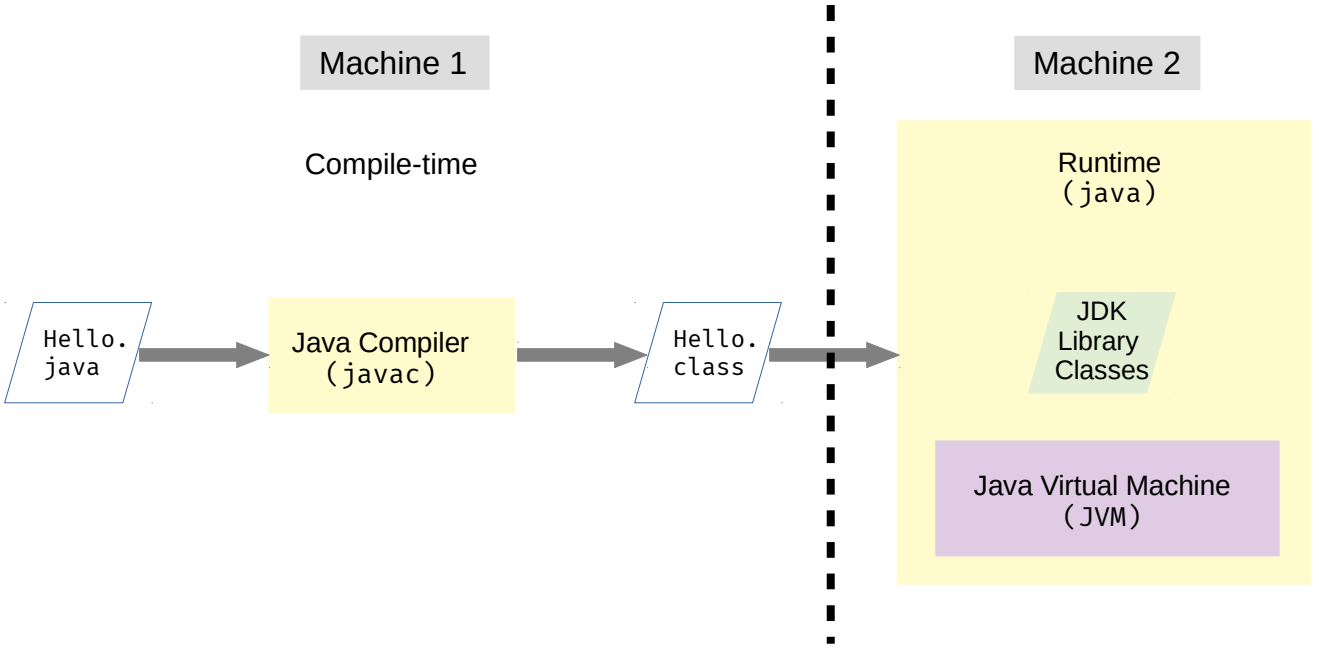
Day 2

- The Java way
- The Java (static) compiler
- JVM architecture
- HotSpot under the hood
- Playing around
- Being a better Java programmer

Homework??

Recall: The Java Compilation+Execution Model

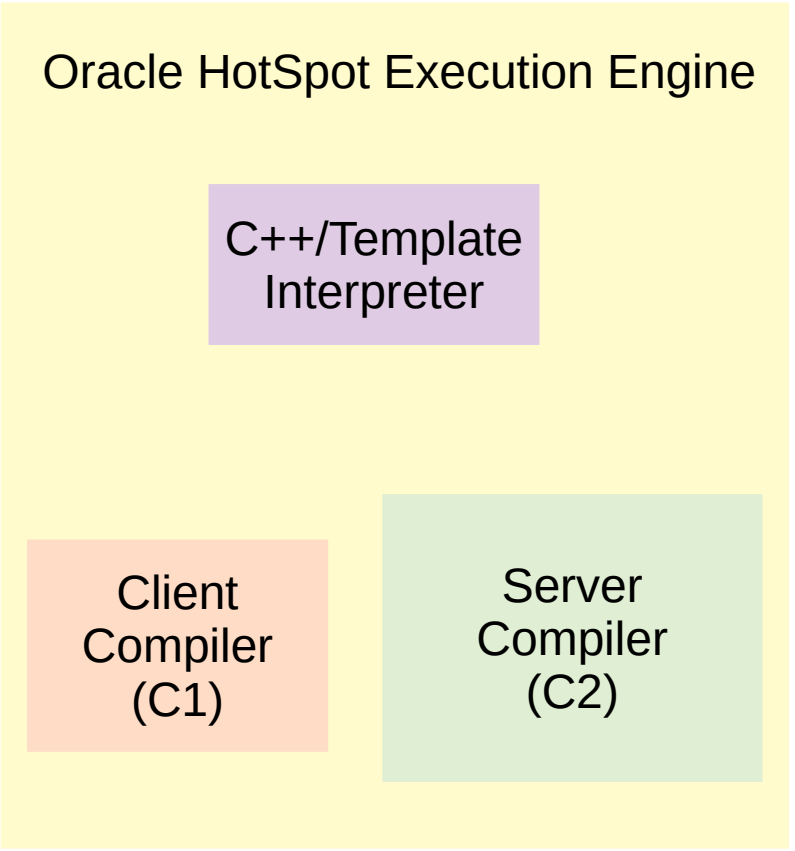




Note the corrections in the above diagram.

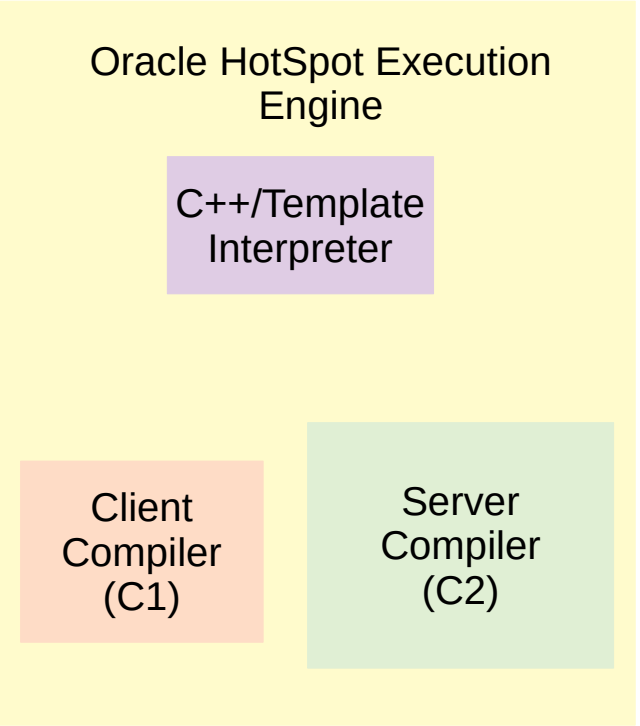
Is Java Bytecode interpreted or compiled?

Java Bytecode is interpreted as well as compiled!!



The “HotSpot” JVM

- HotSpot uses *tiered* compilation
  - Starts off with interpreter
  - Hot spots get compiled as they get executed
- Two interpreters:
  - C++ interpreter (deprecated)
  - Template interpreter
- Just-In-Time (JIT) Compilers:
  - C1 (aka *client*)
  - C2 (aka *server*)



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

Template Interpreter

- Templates of (hardware-specific) assembly code available for each bytecode
- An *interpreter-generator* expands the templates into an address-filled sequence

```
arraylength  190
  0x00000001068fe9a0: pop    %rax
  0x00000001068fe9a1: mov    0xc(%rax),%eax
  0x00000001068fe9a4: movzbl 0x1(%r13),%ebx
  0x00000001068fe9a9: inc    %r13
  0x00000001068fe9ac: movabs $0x106293760,%r10
  0x00000001068fe9b6: jmpq   *(%r10,%rbx,8)
  0x00000001068fe9ba: nopw   0x0(%rax,%rax,1)
```

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

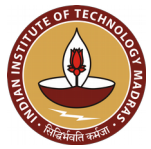
The C2 Compiler

- 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

Performance matters the most

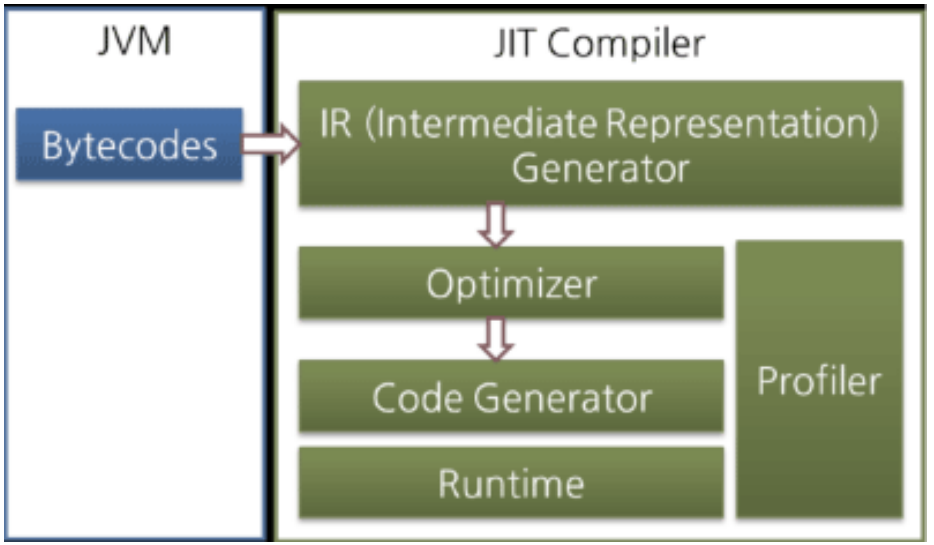
- Template interpreter is fast, but still slower than native code
- Native code runs fast, but generation (compilation) is costlier
- Solution: Adaptive profiling-based JIT compilation
- Wise old saying:

  80% of the execution-time is spent over 20% of the code
- What all to count:
  - Number of times a method is called (invocation count)
  - Number of times a loop is executed (backedge count)
  - And?



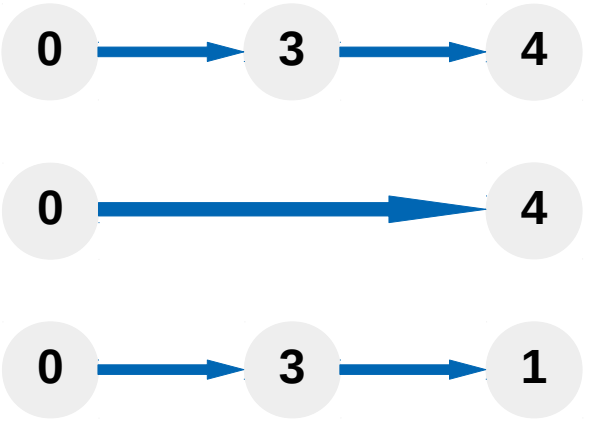
JIT Compilation in the HotSpot JVM

- Hot methods are inserted into a *compilation queue*
- Compiler threads compile methods in the background, while interpretation continues
- Entry points of methods are changed dynamically
- Hot loops are replaced *on-the-stack* (On-Stack Replacement, OSR)



Compilation Levels

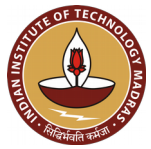
- 0 – Interpreter
- 1 – Pure C1
- 2 – C1 with invocation and backedge counting
- 3 – C1 with full profiling
- 4 – C2 (full optimization)



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

- The Java way
- The Java (static) compiler
- JVM architecture
- HotSpot under the hood
- Playing around
- Being a better Java programmer

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

Some Important Flags: Memory

- Print default values:

```
java -XX:+PrintFlagsFinal | grep StackSize
java -XX:+PrintFlagsFinal | grep HeapSize
```

- Modify heap and stack size:

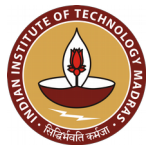
```
-Xss<heap-size>[unit]
-Xms<heap-size>[unit]
-Xmx<heap-size>[unit]
```

- Garbage collection:

```
-verbosegc
-XX:+PrintGCDetails
```

Some Important Flags: Compilation

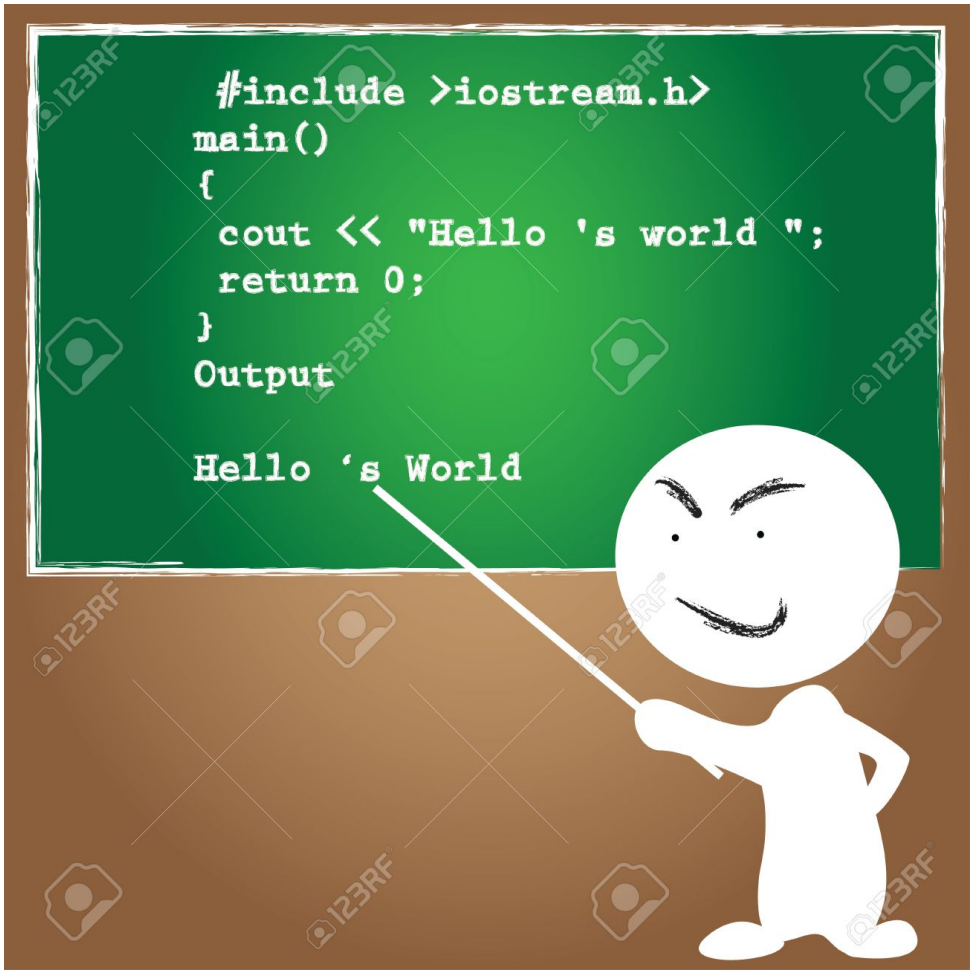
- Compilation details: `-XX:+PrintCompilation`
- Dump assembly: `-XX:+PrintInterpreter`
- Interpreter-only mode: `-Xint`
- Compiler-only mode: `-Xcomp`
- Disable levels 1, 2, and 3: `-XX:-TieredCompilation`
- Stop compilation at level `n`: `-XX:TieredStopAtLevel=n`



Making things even faster

- Intrinsics
  - Implemented directly in native code
  - Common intrinsics:
    - `Thread.currentThread()`
    - `System.arrayCopy()`
    - `System.clone()`
    - `System.nanoTime(), currentTimeMillis()`
    - `String.indexOf()`
    - `Math.*`

Let’s see some source code…

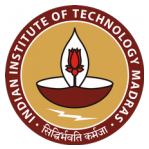


OpenJDK8 Project Structure

```
.
├── build
├── common
├── corba
├── hotspot
├── jaxp
├── jaxws
├── jdk
├── langtools
├── make
├── nashorn
├── nbproject
├── test
└── tmp
```

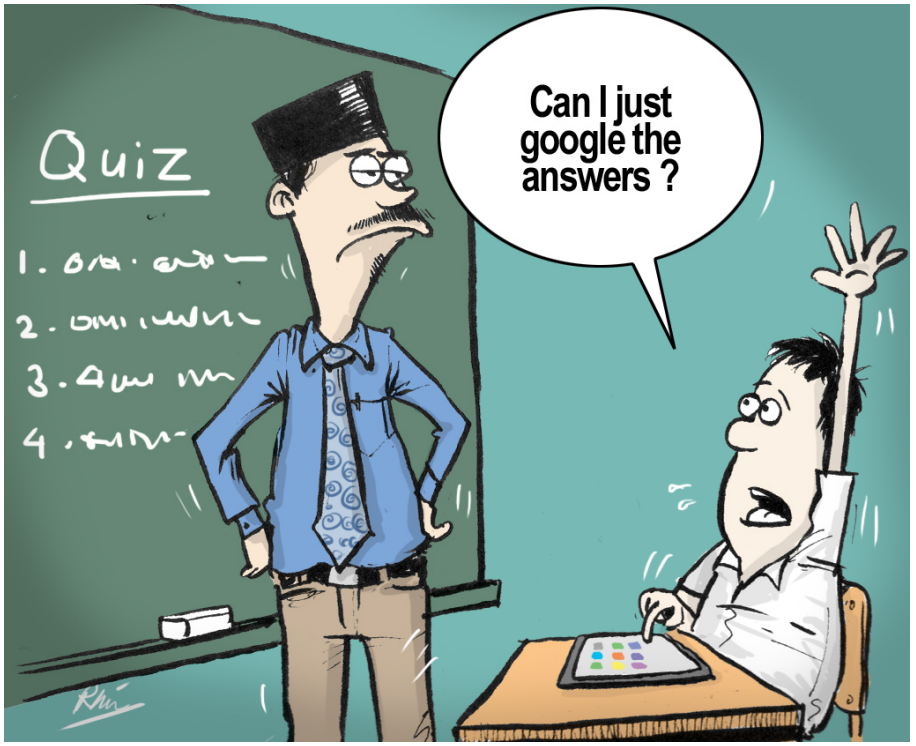
```
jdk/src/share/classes/
├── com
│   ├── oracle
│   └── sun
├── java
│   ├── applet
│   ├── awt
│   ├── beans
│   ├── io
│   ├── lang
│   ├── math
│   ├── net
│   ├── nio
│   ├── rmi
│   ├── security
│   ├── sql
│   ├── text
│   ├── time
│   └── util
└── javax
    ├── accessibility
    ├── crypto
    ├── imageio
    ├── management
    ├── naming
    └── net
```

```
hotspot/src/share/vm/
├── adlc
├── asm
├── c1
├── ci
├── classfile
├── code
├── compiler
├── gc_implementation
├── gc_interface
├── interpreter
├── libadt
├── memory
├── oops
├── opto
├── precompiled
├── prims
├── runtime
├── services
├── shark
├── trace
└── utilities
```



Quiz-2: Which collection should I use?

- Vector or ArrayList?
- Hashtable or HashMap?
- StringBuffer or StringBuilder?



The End is Near

- The Java way
- The Java (static) compiler
- JVM architecture
- HotSpot under the hood
- Playing around
- Being a better Java programmer

Profiling with HPROF

- Comes free with the JRE
- Monitor CPU usage:
  - `java -agentlib:hprof=cpu=samples` *`Klass`*
  - `java -agentlib:hprof=cpu=samples,depth=3`
- Find out time-consuming method(s) and optimize them
- Can be used even when you press CTRL-C
  - Very useful for finding out infinite loops

Some key learnings

- Java programs are not slow.
- Java programs are interpreted *as well as* compiled.
- No need to break encapsulation for performance; most getters and setters get inlined.
- Learn to take advantage of the tools at your disposal (read *javap, hprof*, etc.).
- Trust the JVM, and help it.
  - GC is very sophisticated, but don't allocate objects unnecessarily.
  - JVM provides thread-safe data structures, but use them only when you need.
- Keep experimenting.

Om Poornamadah Poornamidam Poornaat-Poornamudachyate |
Poornasya Poornamaadaaya Poornamevaa-Vashishyate ||

Things we did not cover:

- Java memory model
- Garbage collection
- Java Native Interface (JNI)
- Profiling using JVisualVM
- Visualizing JIT compilation using JITWatch
- Translating lambdas (JDK8)
- Compiler control, AOT compilation (JDK9)
- And much more …

Stay Hungry, Stay Foolish, Stay Connected

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

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

manasthakur17@gmail.com

manasthakur.wordpress.com

linkedin.com/in/manasthakur

manasthakur.github.io/docs/jvm-internals.pdf