How the HotSpot and Graal JVMs Execute Java Code

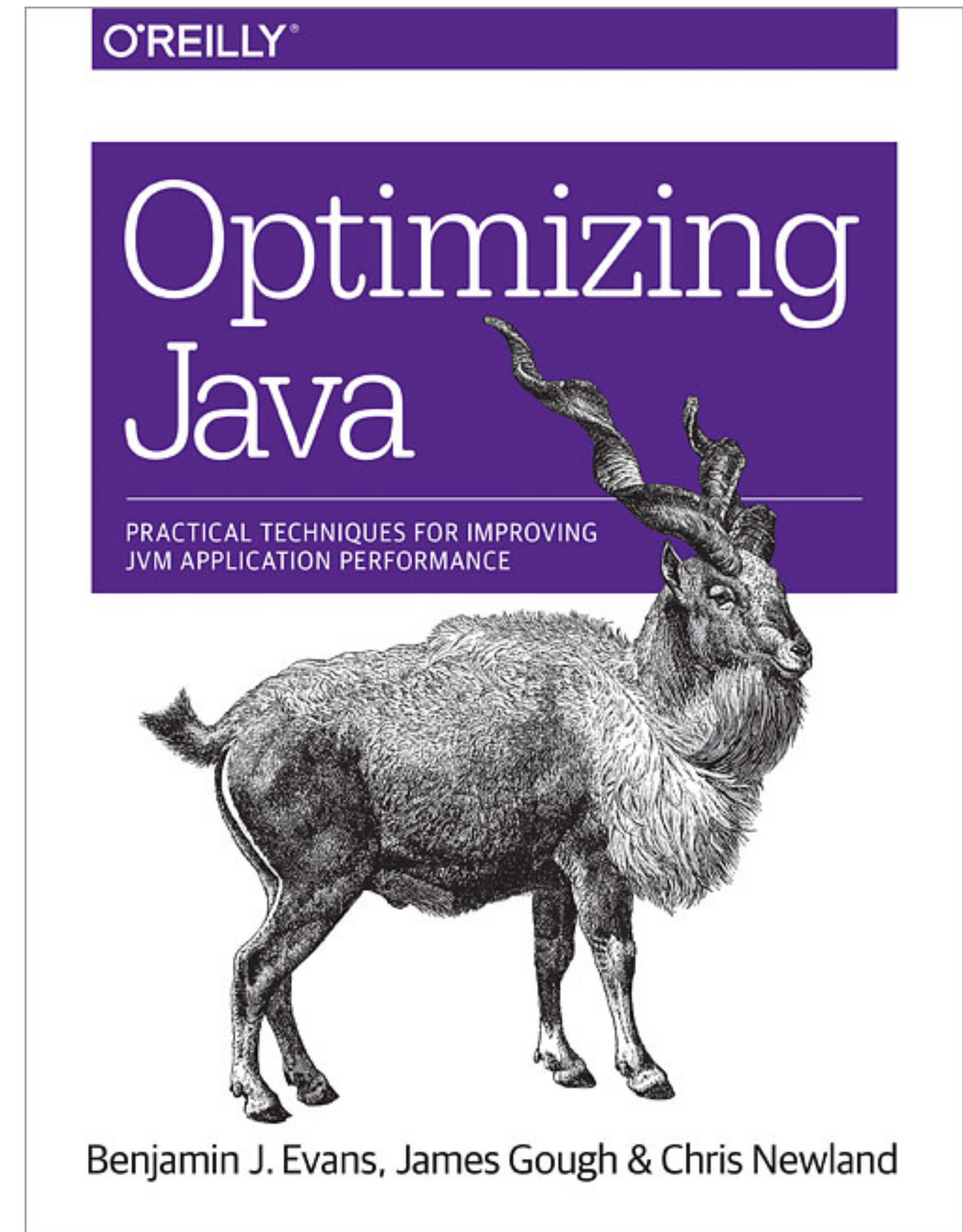
James Gough - **@Jim__Gough**

About Me

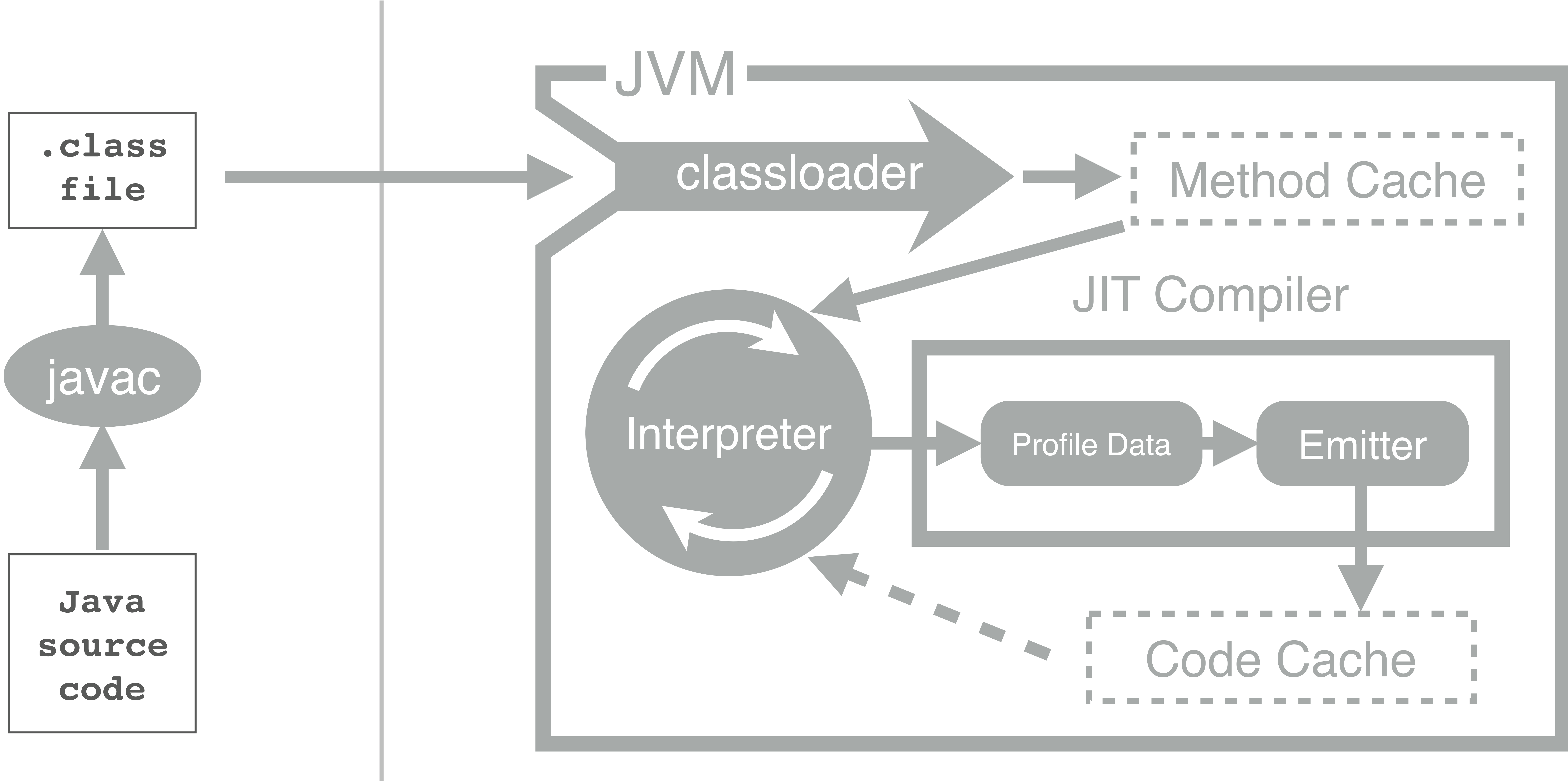
- University of Warwick Graduate
  - Interested in compilers and performance

- London Java Community
- Helped design and test JSR-310 (Date Time)
- Developer and Trainer
  - Teaching Java and C++ to graduates
- Co-Author of Optimizing Java
- Developer on Java based API Gateways
- Occasional Maven Hacker

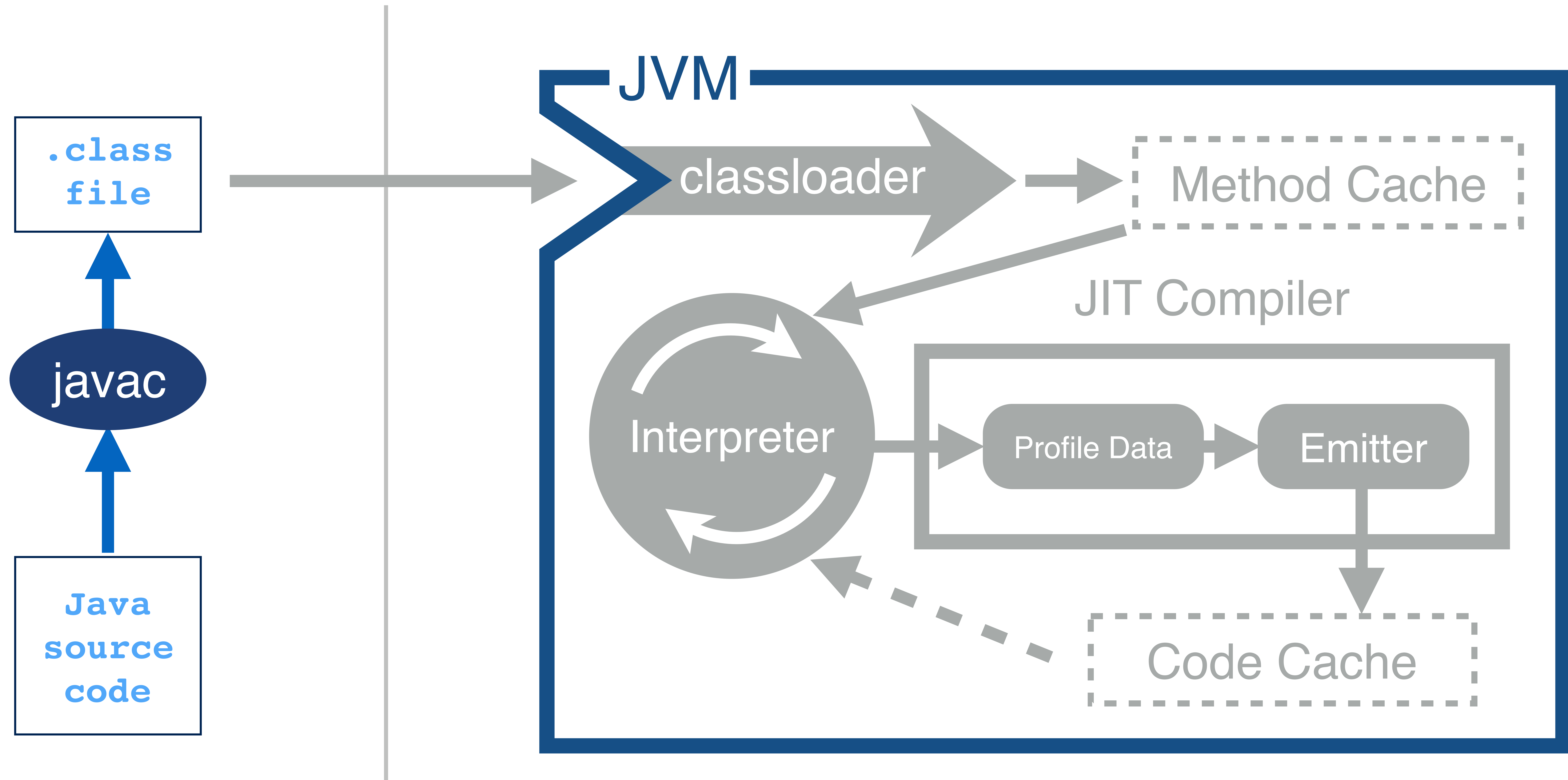
Morgan Stanley



Exploring the JVM



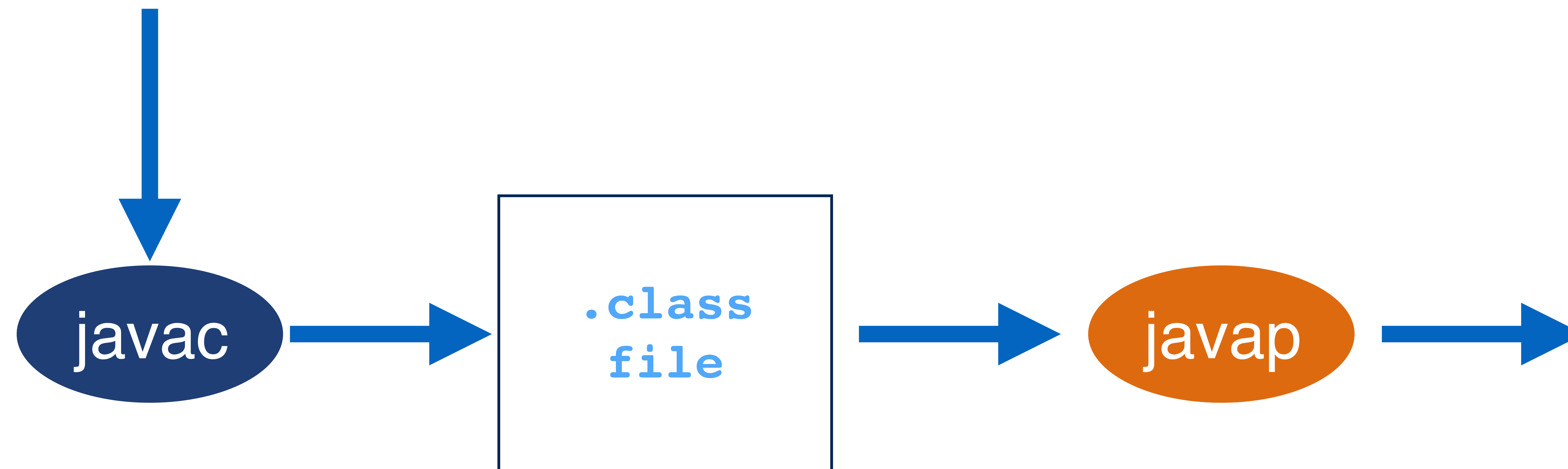
Building Java Applications



A Simple Example

```java
public class HelloWorld {
	public static void main(String[] args) {
		for(int i=0; i < 1_000_000; i++) {
			printInt(i);
		}
	}

	public static void printInt(int number) {
		System.err.println("Hello World" + number);
	}
}
```




```
public HelloWorld();
  Code:
   0: aload_0
   1: invokespecial #1 // Method java/lang/Object."<init>":()V
   4: return

 public static void main(java.lang.String[]);
  Code:
    0: iconst_0
    1: istore_1
    2: iload_1
    3: ldc          #2                  // int 1000000
    5: if_icmpge    18
    8: iload_1
    9: invokestatic  #3                 // Method printInt:(I)V
   12: iinc          1, 1
   15: goto          2
   18: return
```

A Simple Example

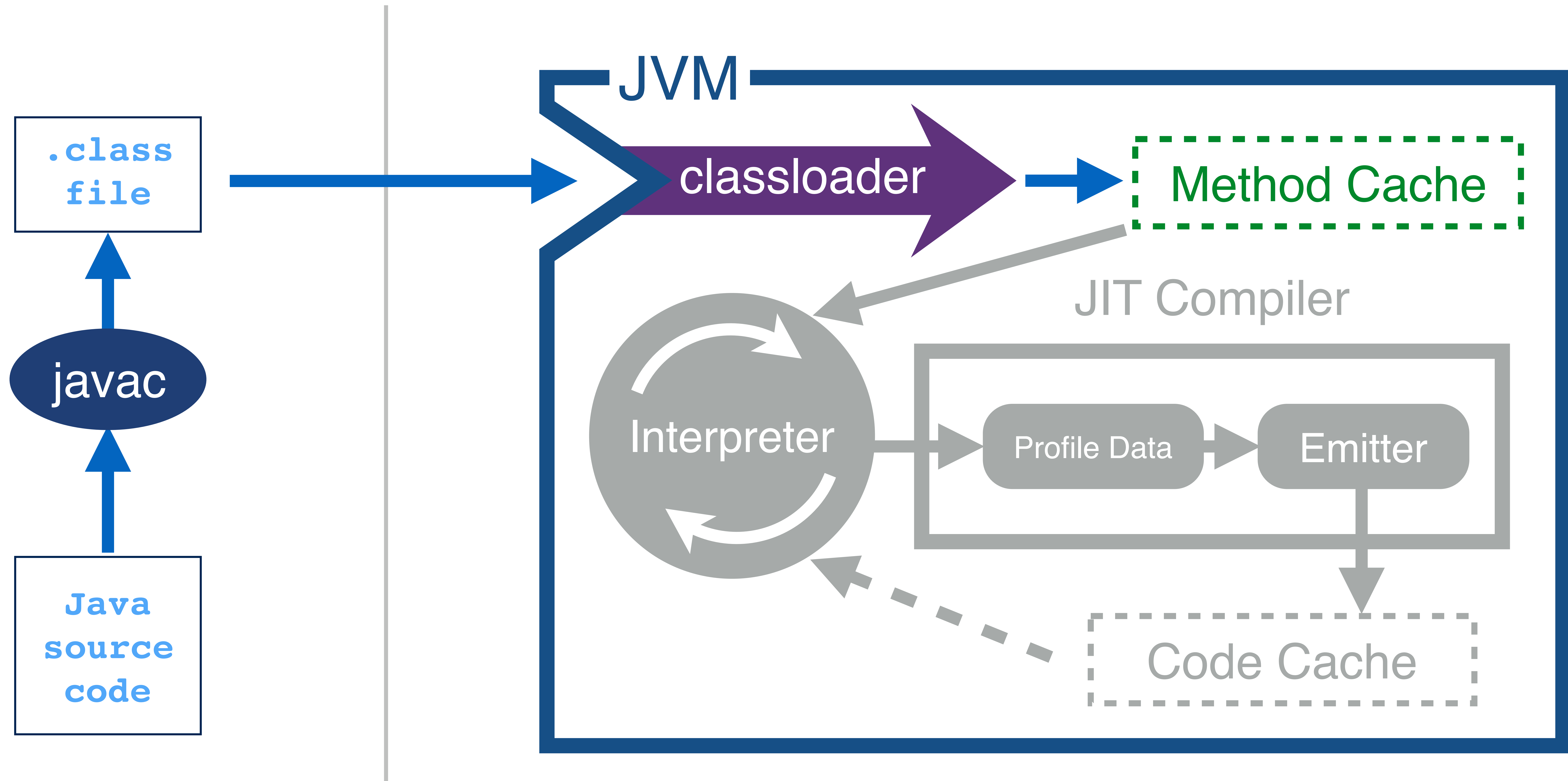
```
public static void printInt(int);
    Code:
       0: getstatic     #4 // Field java/lang/System.err:Ljava/io/PrintStream;
       3: new           #5 // class java/lang/StringBuilder
       6: dup
       7: invokespecial #6 // Method java/lang/StringBuilder."<init>":()V
      10: ldc           #7 // String Hello World
      12: invokevirtual #8
// Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

Java 8

```
public static void printInt(int);
    Code:
       0: getstatic     #4
// Field java/lang/System.err:Ljava/io/PrintStream;
       3: iload_0
       4: invokedynamic #5,  0
// InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;
       9: invokevirtual #6
// Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: return
```

Java 9+

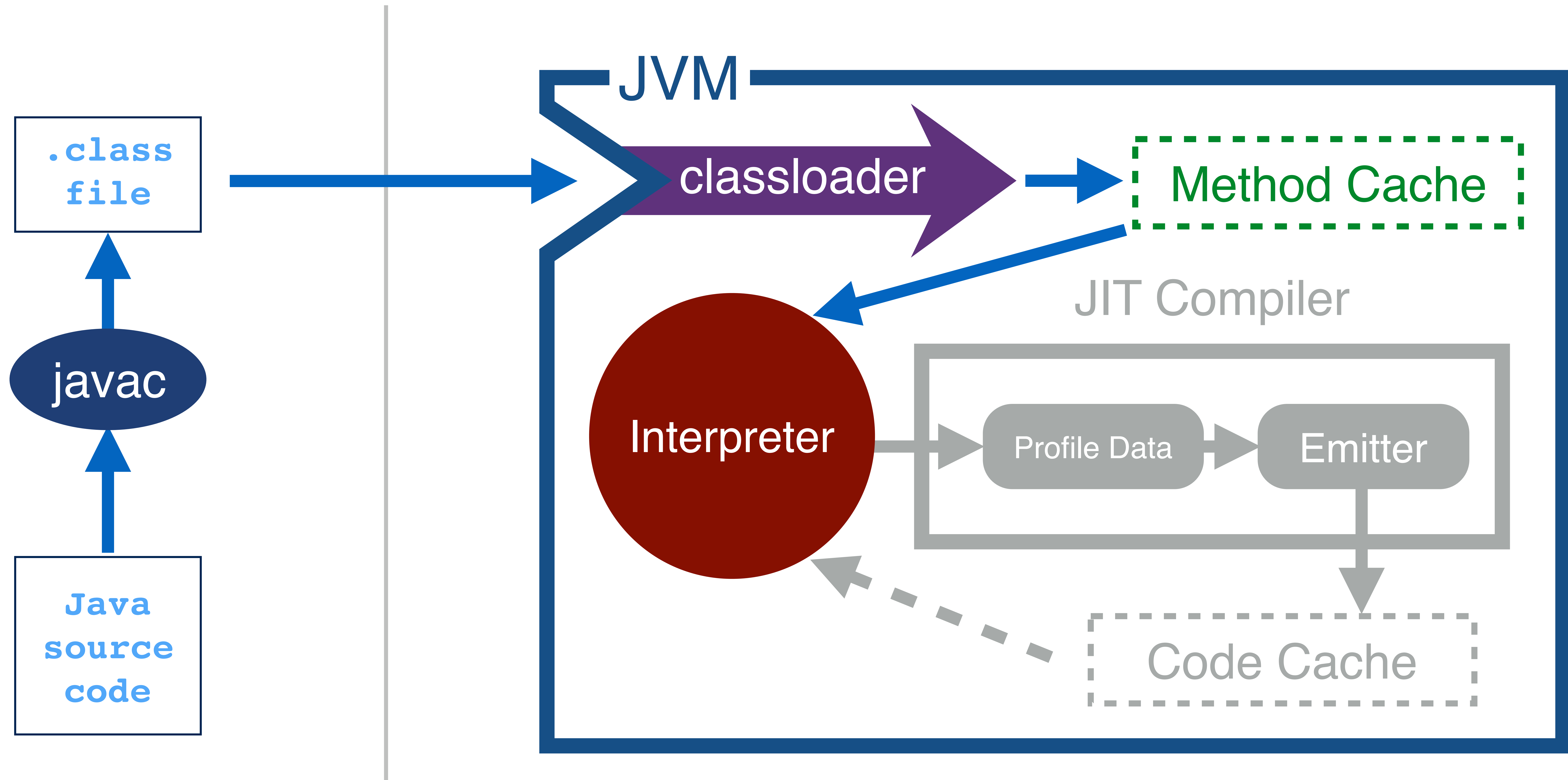
Classloaders



Classloaders

- Classes are loaded just before they are needed
    - Proven by the painful ClassNotFoundException, NoClassDefFoundError
    - Build tools hide this problem away from us
- Maps class file contents into the JVM `klass` object
    - Instance Methods are held in the `klassVtable`
    - Static variables are held in the `instanceKlass`
- You can write your own classloader to experiment
    - https://github.com/jpgough/watching-classloader

Interpreting Bytecode

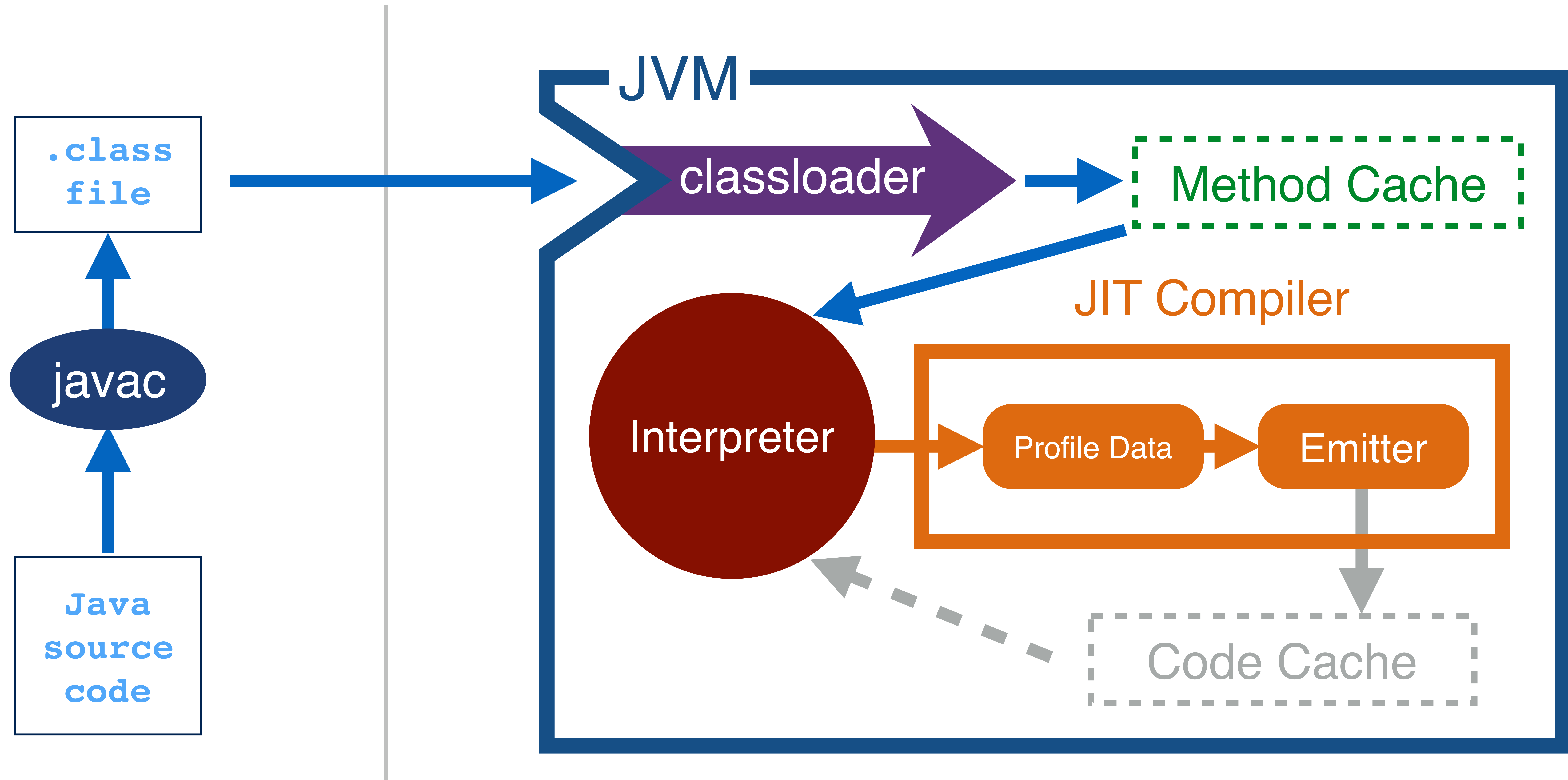


Interpreting Bytecode

- Bytecode initially fully interpreted
- Conversion of instructions to machine instructions
    - Using template interpreter
- Time not spent compiling code that is only used once
- Allows the JVM to establish "true profile" of the application

https://speakerdeck.com/alblue/javaone-2016-hotspot-under-the-hood?slide=21

Just in Time Compilation (JIT)



The HotSpot Compiler

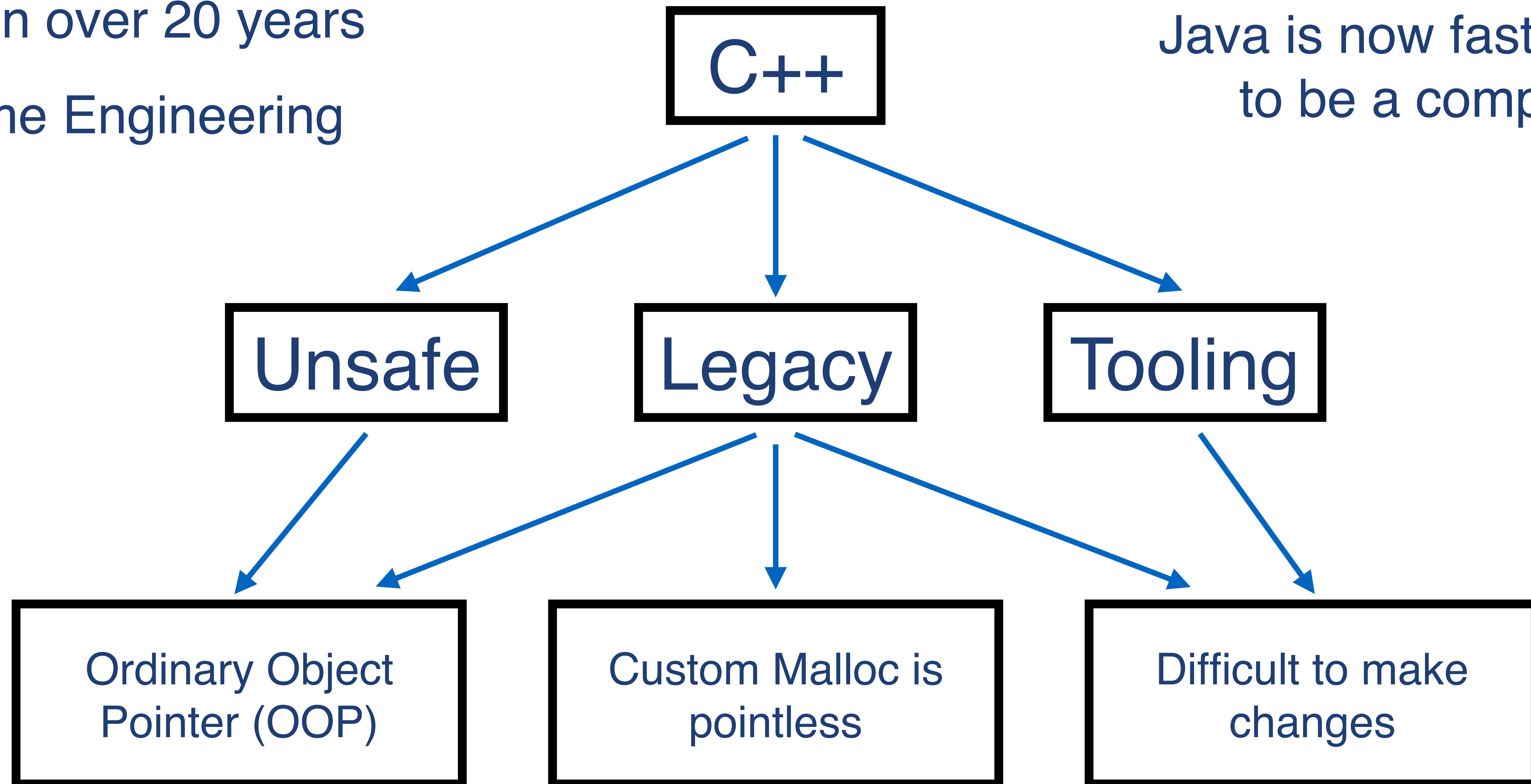
- Java observes code executing using profile data
- Triggers a compilation request on meeting the threshold
  - Startup methods may only be invoked a few times
- Utilising the profile enables informed optimisation
  - Classloaders mean then JVM doesn't know what will run
- Emits machine code to replace interpreted bytecode
- C2 is the main HotSpot Compiler implemented in C++

Challenges with the C2 Compiler

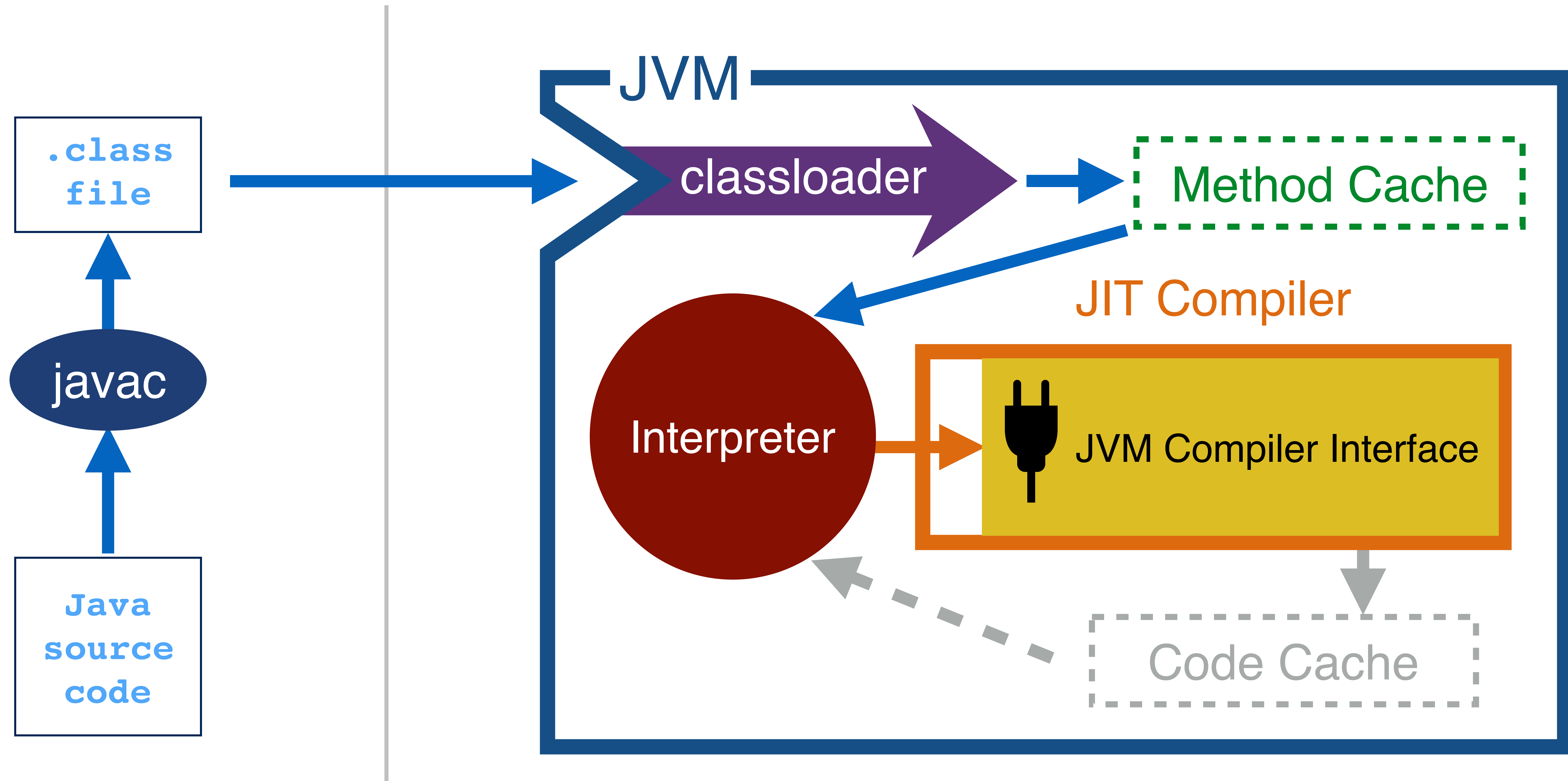
Built upon over 20 years

Awesome Engineering

Java is now fast enough to be a compiler?



Evolving the JIT Compiler



JVM Compiler Interface (JVMCI)

- Provides access to VM structures for compiler
  - Fields, methods, profile information…
- Mechanism to install the compiled code
  - Along with metadata required for GC and deoptimization
- Produce machine code at a method level
- Uses JEP-261 (Modules) to protect and isolate

Graal as a JIT

- A JIT compiler has a simple series of inputs
  - Method to compile to convert bytecode to assembly
  - A graph of nodes to convey the structure and context
  - The profile of the running application
- Implementing a JIT in Java is quite compelling
  - Language level safety in expressions
  - Easy to debug, great tools and IDE support

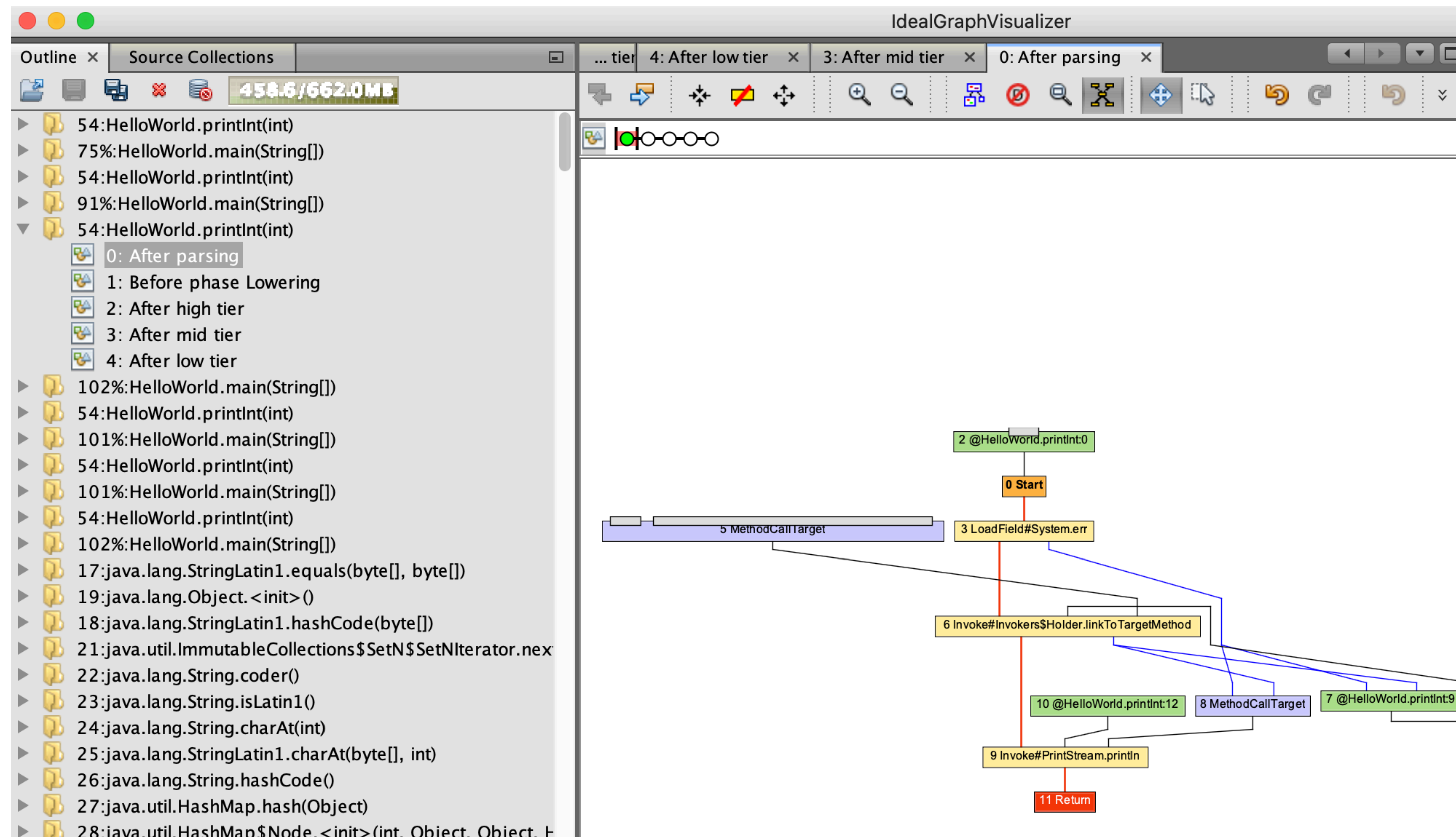
Getting Started with Graal

- `mx` command line tool for graal
- pull in the graalvm project (work within graal/compiler)
- `mx build` to build our local compiler `mx ideinit`
- `mx -d vm` (debug and install our local suite as the vm)

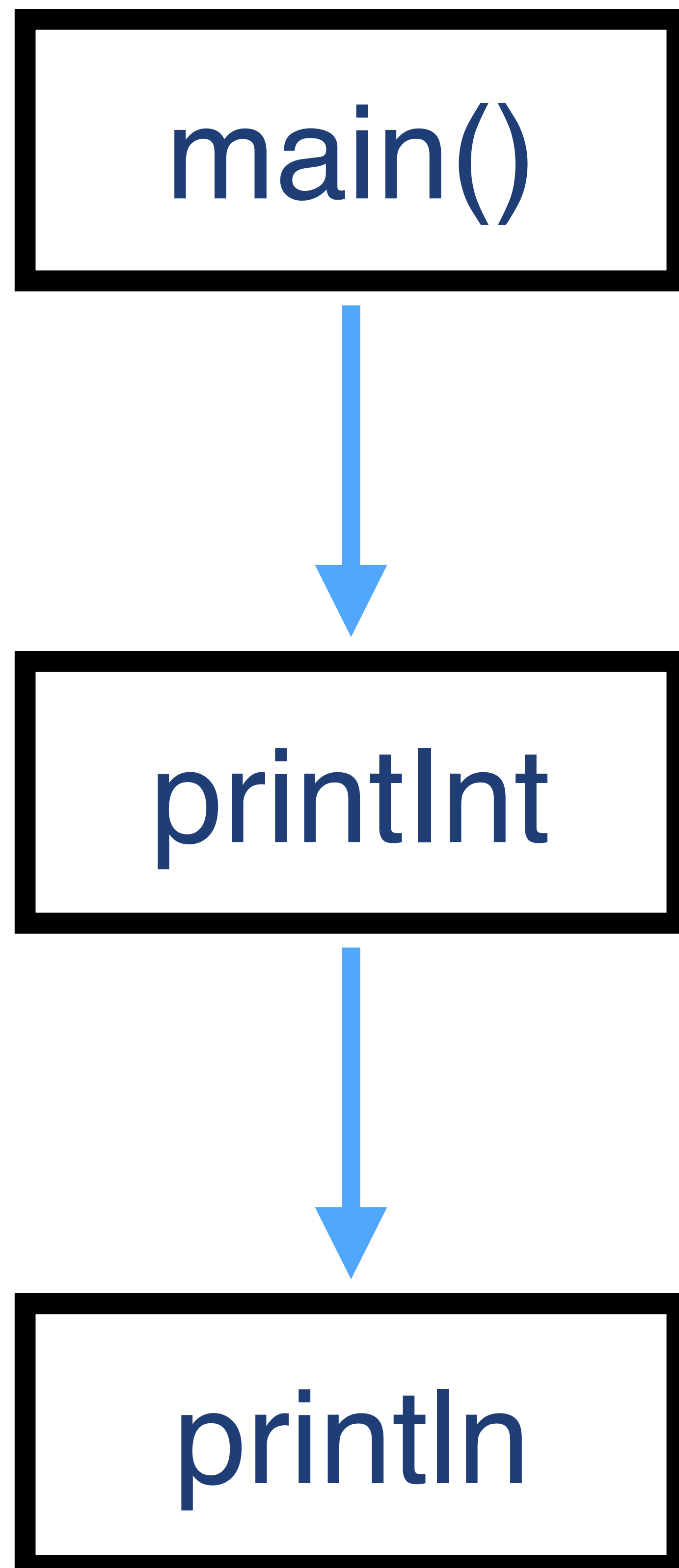
```
-XX:+UnlockExperimentalVMOptions
-XX:+EnableJVMCI
-XX:+UseJVMCICompiler
-XX:-TieredCompilation
-XX:CompileOnly=HelloWorld,System/err/println
-Dgraal.Dump
HelloWorld
```

Getting Started with Graal

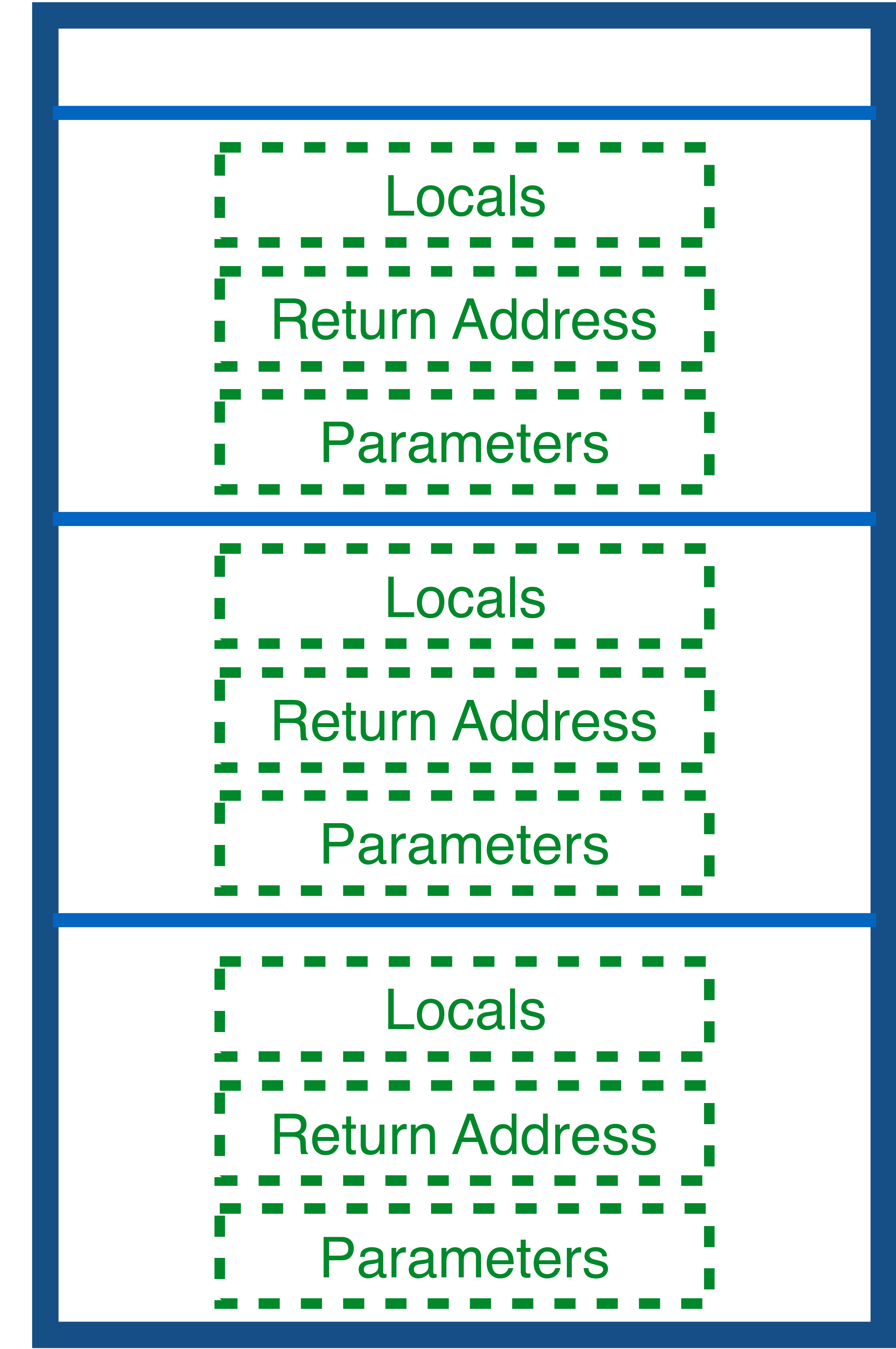
- IdealGraphVisualizer - Oracle Enterprise tool



Exploring Inlining



Frame Pointer



Live Debug

org.graalvm.compiler.phases 〉 src 〉 org 〉 graalvm 〉 compiler 〉 phases 〉 PhaseSuite

InliningData.java | InliningPhase.java | BasePhase.java | PhaseSuite.java | HotSpotGraalCompiler.java | StructuredGraph.java | CompilationRequest.cla

Project: jdk, mx, org, graalvm, compiler, api, truffle, org.graalvm.compiler.asm.am, org.graalvm.compiler.code, org.graalvm.compiler.core, org.graalvm.compiler.core.aa, org.graalvm.compiler.core.am, org.graalvm.compiler.core.co, org.graalvm.compiler.core.llv, org.graalvm.compiler.core.sp, org.graalvm.compiler.core.tes, org.graalvm.compiler.debug, org.graalvm.compiler.graph, org.graalvm.compiler.graph.t, org.graalvm.compiler.hotspot, org.graalvm.compiler.hotspo, bin

```
199                     return true;
200                 }
201             }
202         }
203         return false;
204     }
205
206     @Override
207     protected void run(StructuredGraph graph, C context) {  graph: "StructuredGraph:1{HotSpotMethod<HelloWorld.printInt(int)>}"
208         for (BasePhase<? super C> phase : phases) {  phase: DeadCodeEliminationPhase@2588
209             phase.apply(graph, context);  phase: DeadCodeEliminationPhase@2588  graph: "StructuredGraph:1{HotSpotMethod<HelloWor
210         }
211     }
212
213     public PhaseSuite<C> copy() {
214         PhaseSuite<C> suite = new PhaseSuite<>();
215         suite.phases.addAll(phases);
216         return suite;
217     }
218 }
219
```

PhaseSuite › run()

Debug: GraalDebug

Debugger | Console

Frames

"JVMCI Co...3: RUNNING

- run:209, PhaseSuite *(org.graalvm.compiler.p*
- apply:209, BasePhase *(org.graalvm.compiler.*
- apply:146, BasePhase *(org.graalvm.compiler.*
- emitFrontEnd:218, GraalCompiler *(org.graalv*
- compile:144, GraalCompiler *(org.graalvm.co*
- compileGraph:129, GraalCompiler *(org.graal*
- compileHelper:228, HotSpotGraalCompiler *(*
- compile:246, HotSpotGraalCompiler *(org.gra*
- performCompilation:170, CompilationTask$
- performCompilation:84, CompilationTask$H

Variables

- this = {HighTier@2573}
- graph = {StructuredGraph@2432} "StructuredGraph:1{HotSpotMethod<HelloWorld.printInt(int)>}"
- context = {HighTierContext@2814}
- phase = {DeadCodeEliminationPhase@2588}
- phases = {ArrayList@2576} size = 11

Watches | Overhead

No watches

Compilation Tiers and Phases

High Tier

1. CanonicalizerPhase
2. **InliningPhase**
3. **DeadCodeEliminationPhase**
4. IncrementalCanonicalizerPhase
5. IterativeConditionalEliminationPhase
6. **LoopFullUnrollPhase**
7. IncrementalCanonicalizerPhase
8. IncrementalCanonicalizerPhase
9. **PartialEscapePhase**
10. EarlyReadEliminationPhase
11. LoweringPhase

Mid Tier

1. LockEliminationPhase
2. IncrementalCanonicalizerPhase
3. IterativeConditionalEliminationPhase
4. LoopSafepointEliminationPhase
5. GuardLoweringPhase
6. IncrementalCanonicalizerPhase
7. LoopSafepointInsertionPhase
8. LoweringPhase
9. OptimizeDivPhase
10. FrameStateAssignmentPhase
11. LoopPartialUnrollPhase
12. ReassociateInvariantPhase
13. DeoptimizationGroupingPhase
14. CanonicalizerPhase
15. WriteBarrierAdditionPhase

Low Tier

1. LoweringPhase
2. ExpandLogicPhase
3. FixReadsPhase
4. CanonicalizerPhase
5. AddressLoweringPhase
6. UseTrappingNullChecksPhase
7. **DeadCodeEliminationPhase**
8. PropagateDeoptimizeProbabilityPhase
9. InsertMembarsPhase
10. SchedulePhase

Dead Code Elimination

- Removes code that is never executed
  - Shrinks the size of the program
  - Avoids executing irrelevant operations
- Dynamic dead code elimination
  - Eliminated base on possible set of values
  - Determined at runtime

Loop Unrolling

- Iteration requires back branches and branch prediction
- For int, char and short loops loop can be unrolled
- Can remove safe point checks
- Reduces the work needed by each “iteration”

Loop Unrolling

```
@Benchmark
public long intStride() {
  long sum = 0;
  for (int i = 0; i < MAX; i++) {
    sum += data[i];
  }
  return sum;
}
```

```
@Benchmark
public long longStride() {
  long sum = 0;
  for (long l = 0; l < MAX; l++) {
    sum += data[(int) l];
  }
  return sum;
}
```

Benchmark	Mode	Cnt	Score		Error	Units
LoopUnrollingCounter.intStride	thrpt	200	**2423.818**	±	2.547	ops/s
LoopUnrollingCounter.longStride	thrpt	200	**1469.833**	±	0.721	ops/s

Excerpt From: Benjamin J. Evans, James Gough, and Chris Newland. “Optimizing Java.”.

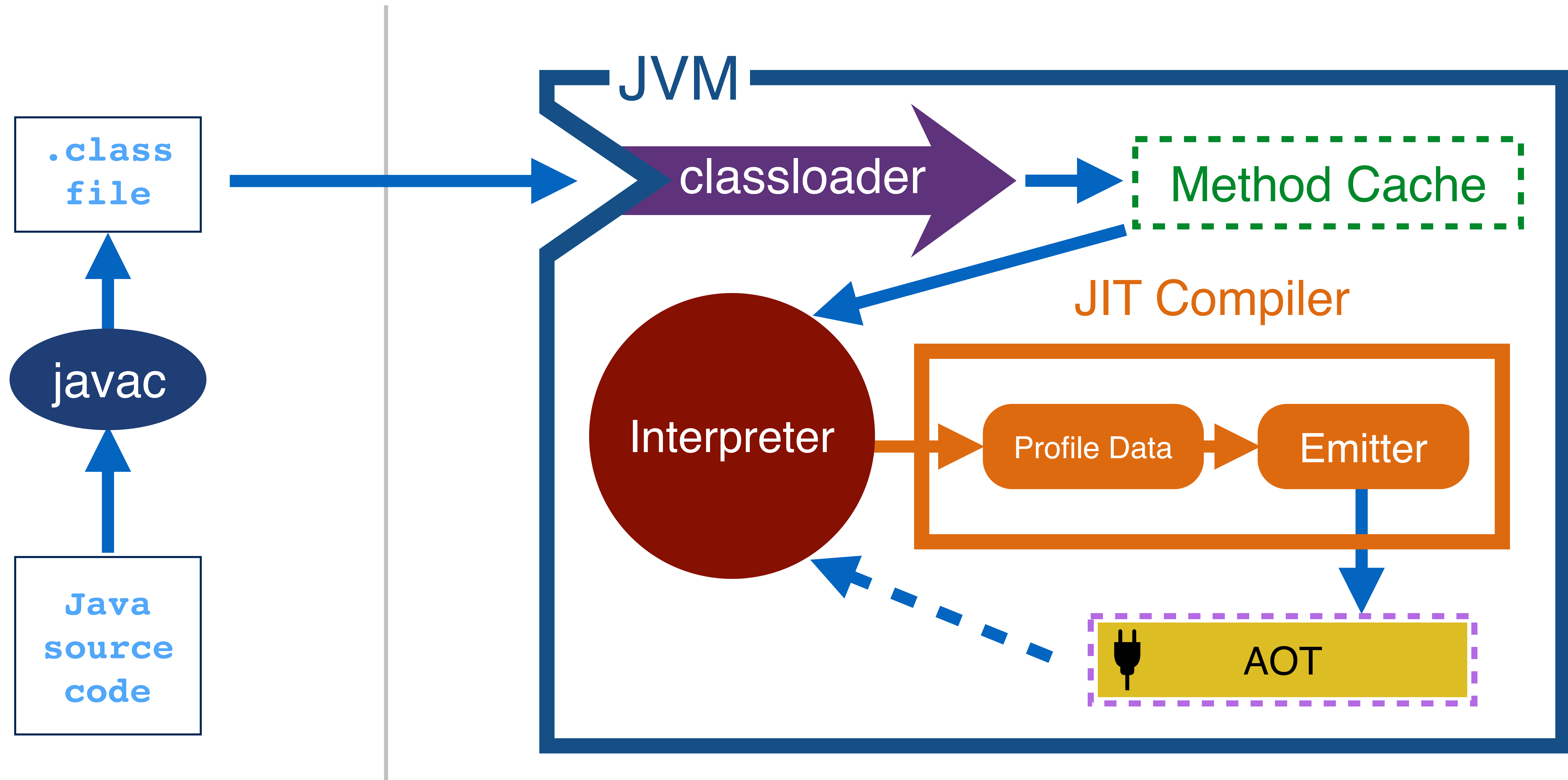
Escape Analysis

- Introduced in later versions of Java 6
- Analyses code to assert if an object
  - Returns or leaves the scope of the method
  - Stored in global variables
- Allocates unescaped objects on the stack
  - Avoids the cost of garbage collection
  - Prevents workload pressures on Eden
  - Beneficial effects to counter high infant mortality GC impact

Monomorphic Dispatch

- When HotSpot encounters a virtual call site, often only one type will ever be seen there
  - e.g. There's only one implementing class for an interface
- Hotspot can optimise vtable lookup
  - Subclasses have the same vtable structure as their parent
  - Hotspot can collapse the child into the parent
- Classloading tricks can invalidate monomorphic dispatch
  - The class word in the header is checked
  - If changed then this optimisation is backed out

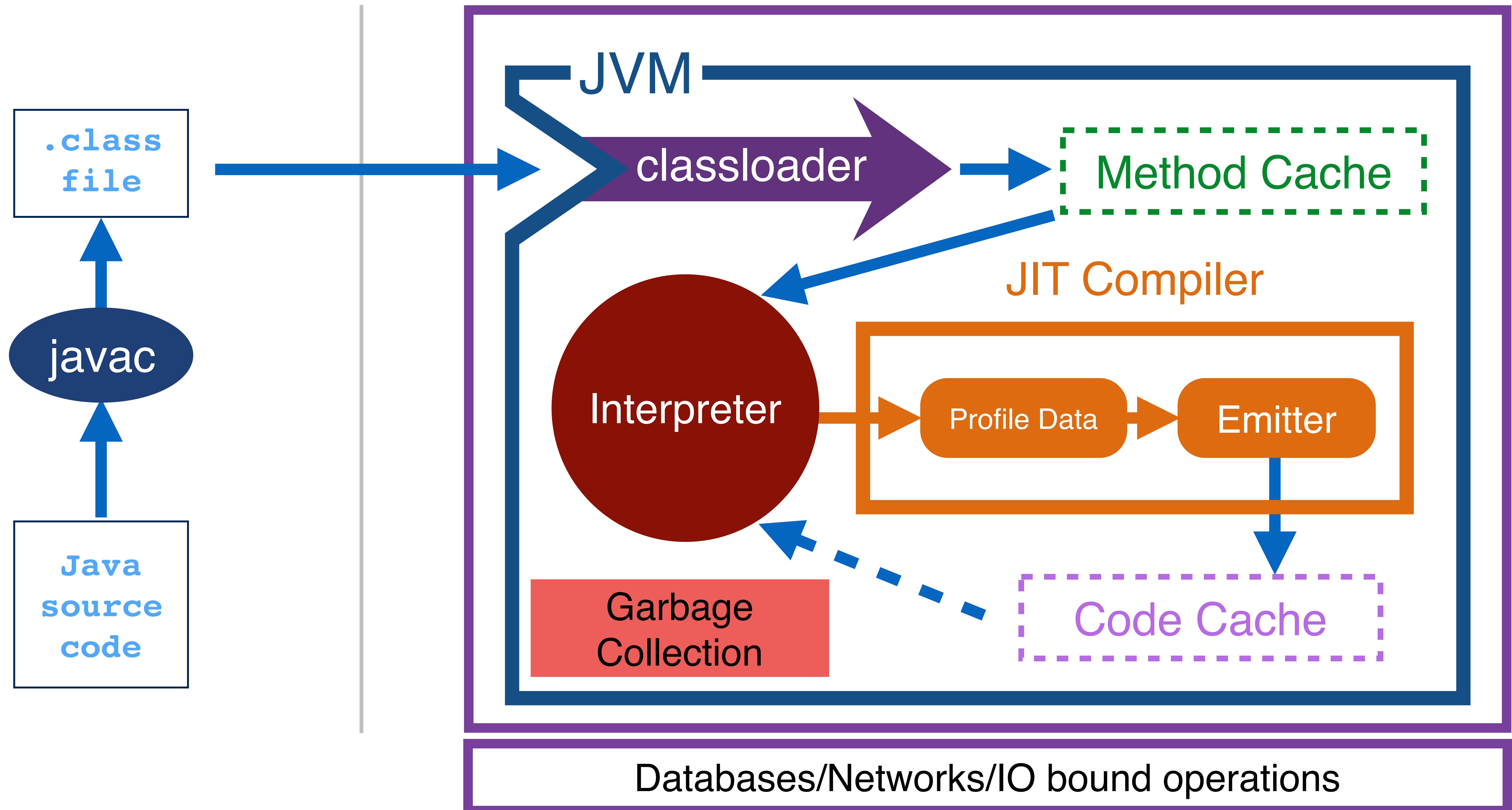
Code Cache



Ahead of Time Compilation

- Achieved using a new tool called `jaotc`
- Graal is used as the code generating backend
- JVM treats AOT code as an extension to the code cache
- JVM must reject incompatible code
  - Fingerprinting techniques are used
- `jaotc --output libHelloWorld.so HelloWorld.class`
- `java -XX:+UnlockExperimentalVMOptions -XX:AOTLibrary=./libHelloWorld.so HelloWorld`

The Bigger Picture



Acknowledgements

- Chris Seaton for his excellent initial post on Graal as a JIT
- Ben Evans for his education, patience and friendship
- Anna Evans for some of the amazing slide graphics
- Martijn Verburg for encouragement and support
- GraalVM and OpenJDK team for the projects
- Alex Blewitt for talk review and HotSpot Under the Hood talk
- NY Java SIG for hosting trial run

Thanks for Listening

James Gough - **@Jim__Gough**