THE PERFORMANCE ENGINEER'S GUIDE TO HOTSPOT JIT COMPILATION

Monica Beckwith

CODE KARAM LLC

QCon NY

Jun 14th 2016

About Me …

Java/JVM/GC Performance Consultant

- Worked with JIT compiler, JVM heuristics, Heap Management, Various Garbage Collectors (GCs)
- Worked with AMD, Oracle and Sun
- Was the Performance Lead for G1 GC

www.codekaram.com
https://www.linkedin.com/in/monicabeckwith
Tweet @mon_beck

Topics We May Cover Today

- HotSpot Virtual Machine
  - Runtime Goal
  - Common Techniques
- Adaptive Optimization - The Plot
  - C1, C2 & Tiered Compilations

©2016 CodeKaram

Topics We May Cover Today

- Adaptive Optimization - The Plot
  - Dynamic Deoptimization
  - Inlining
  - Intrinsics
  - Vectorization (Auto and otherwise)
  - Escape Analysis

©2016 CodeKaram

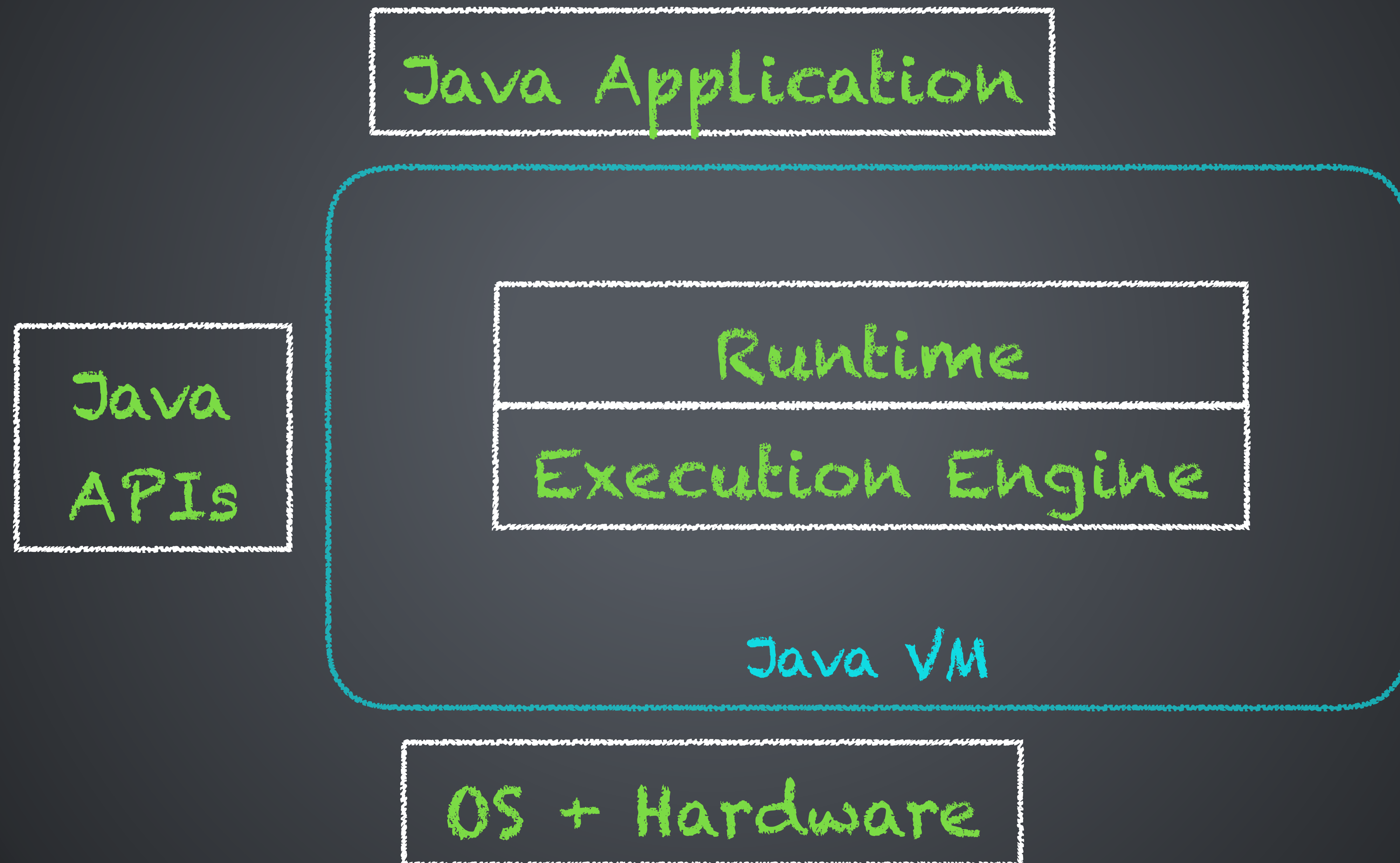
Topics We May Cover Today

- Time Check!
- Adaptive Optimization - The Plot
  - Compressed Oops
  - Compressed Class Pointers
- Q & A

©2016 CodeKaram

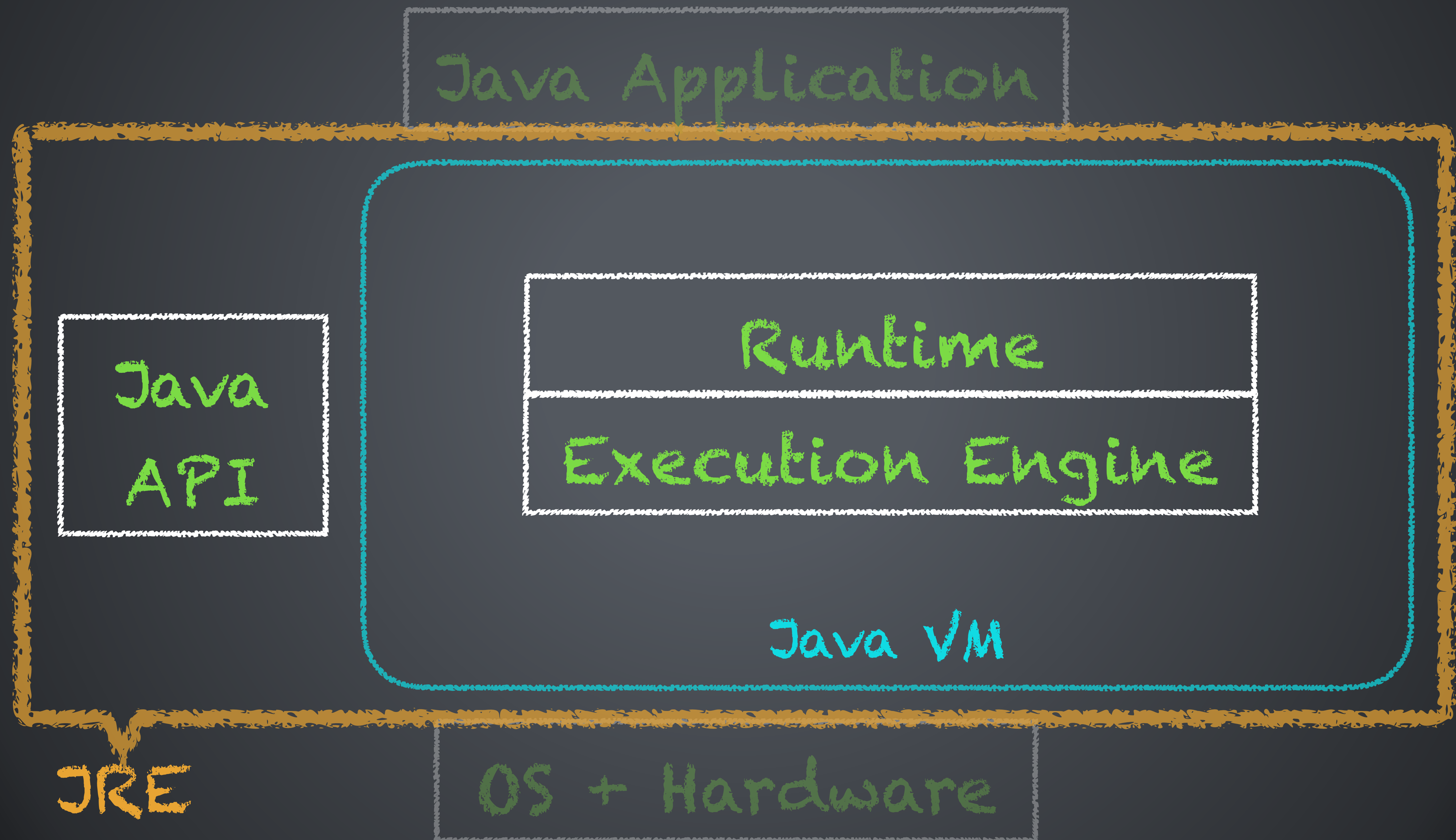
HotSpot Virtual Machine

Let's Peek Under The Hood



©2016 CodeKaram

Java Runtime Environment

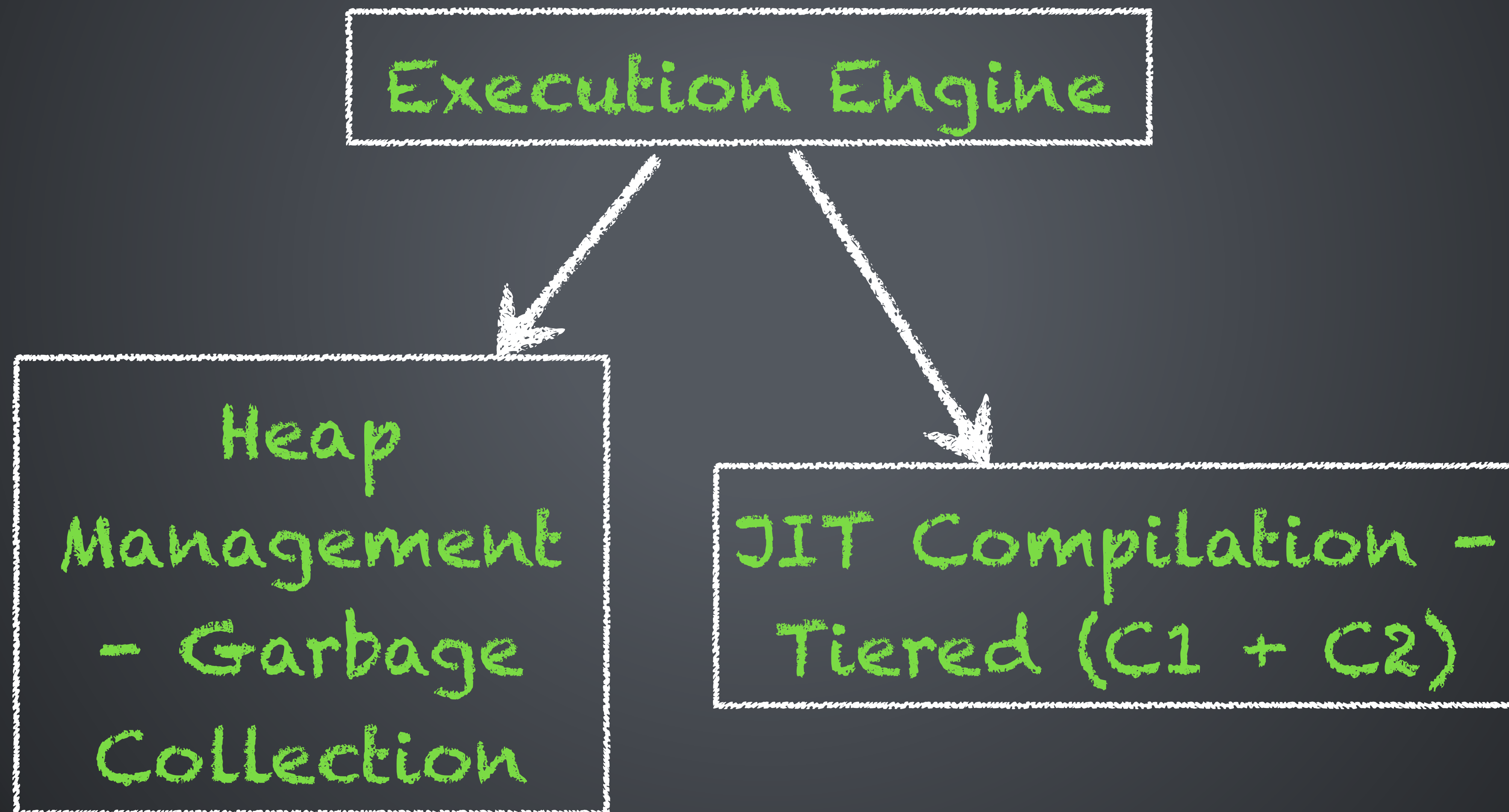


©2016 CodeKaram

The Helpers

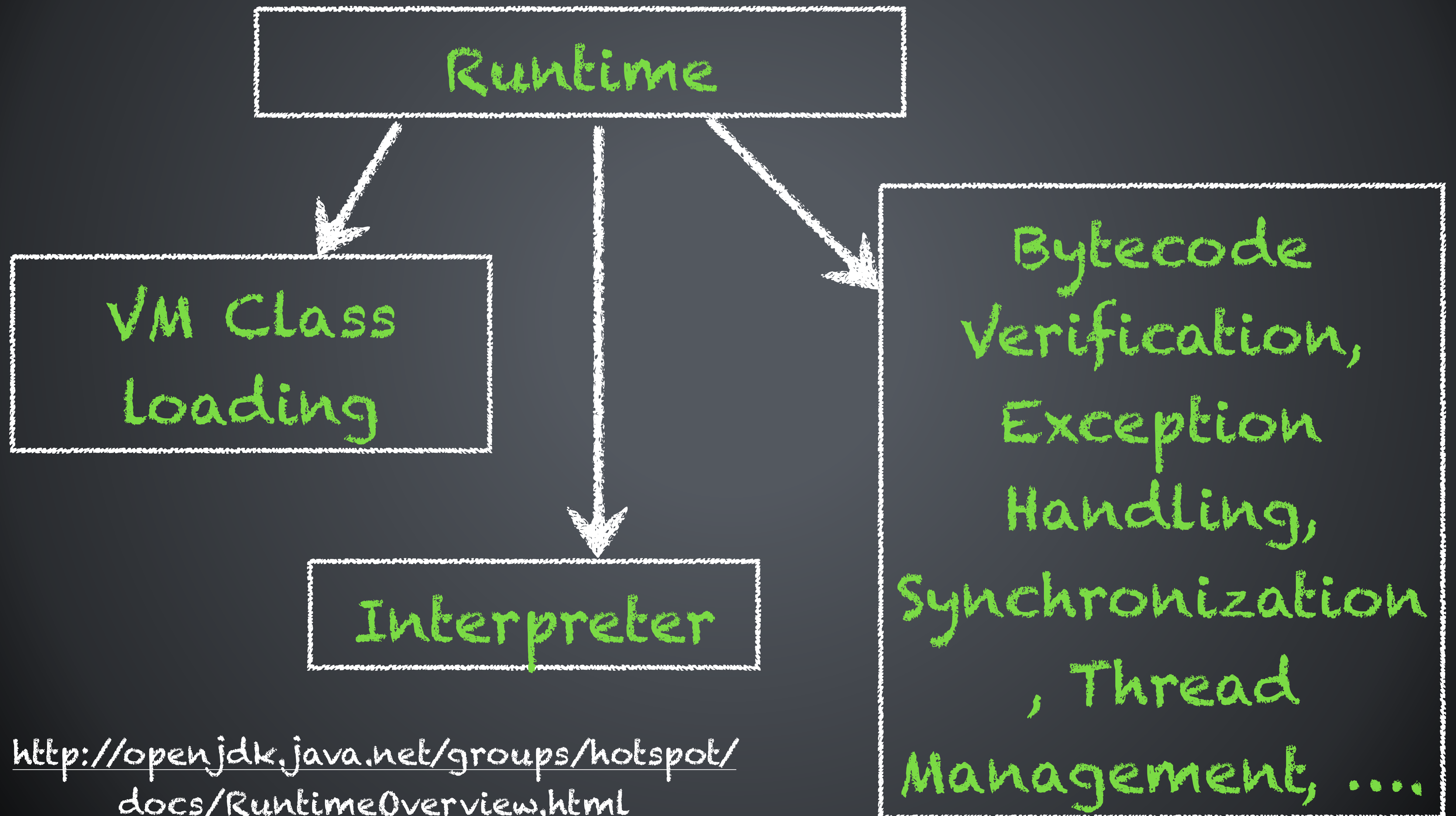
©2016 CodeKaram

HotSpot VM



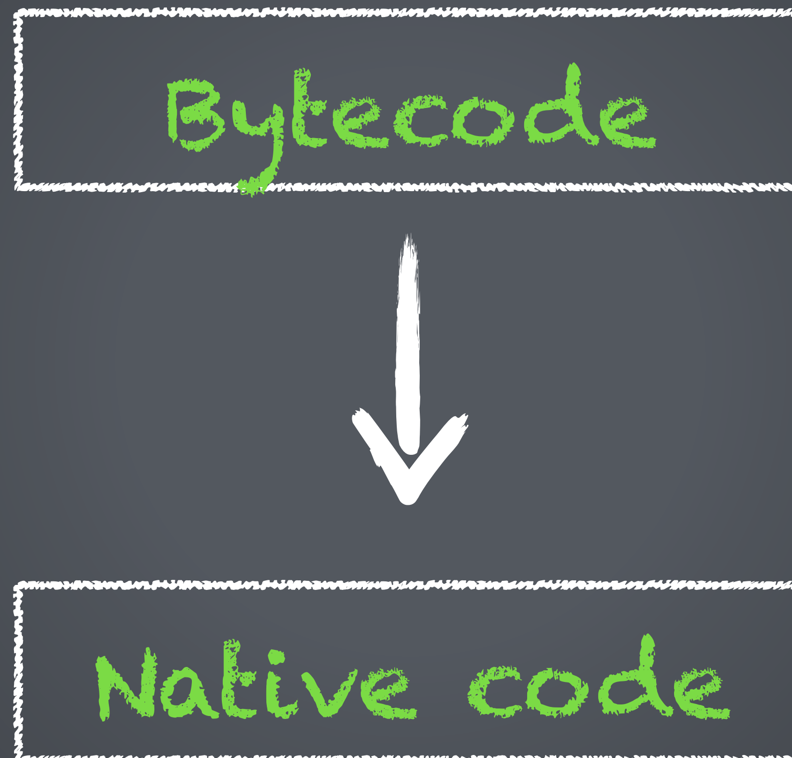
©2016 CodeKaram

HotSpot VM



©2016 CodeKaram

Runtime Goal



©2016 CodeKaram

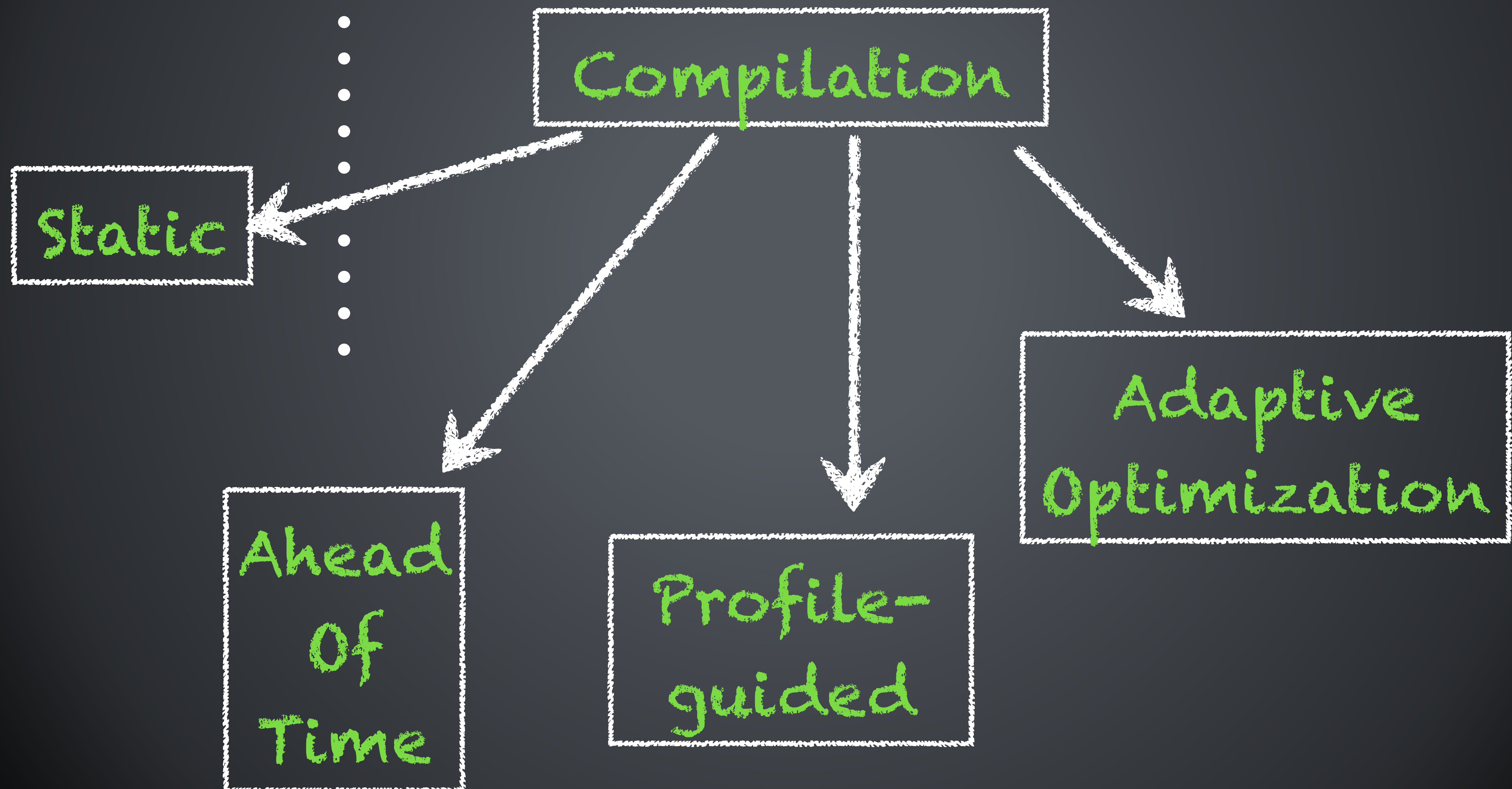
HotSpot VM - Template Interpreter

TemplateTable

Bytecode	TemplateTable
Bytecode 1	Template Native Code 1
Bytecode 2	Template Native Code 2

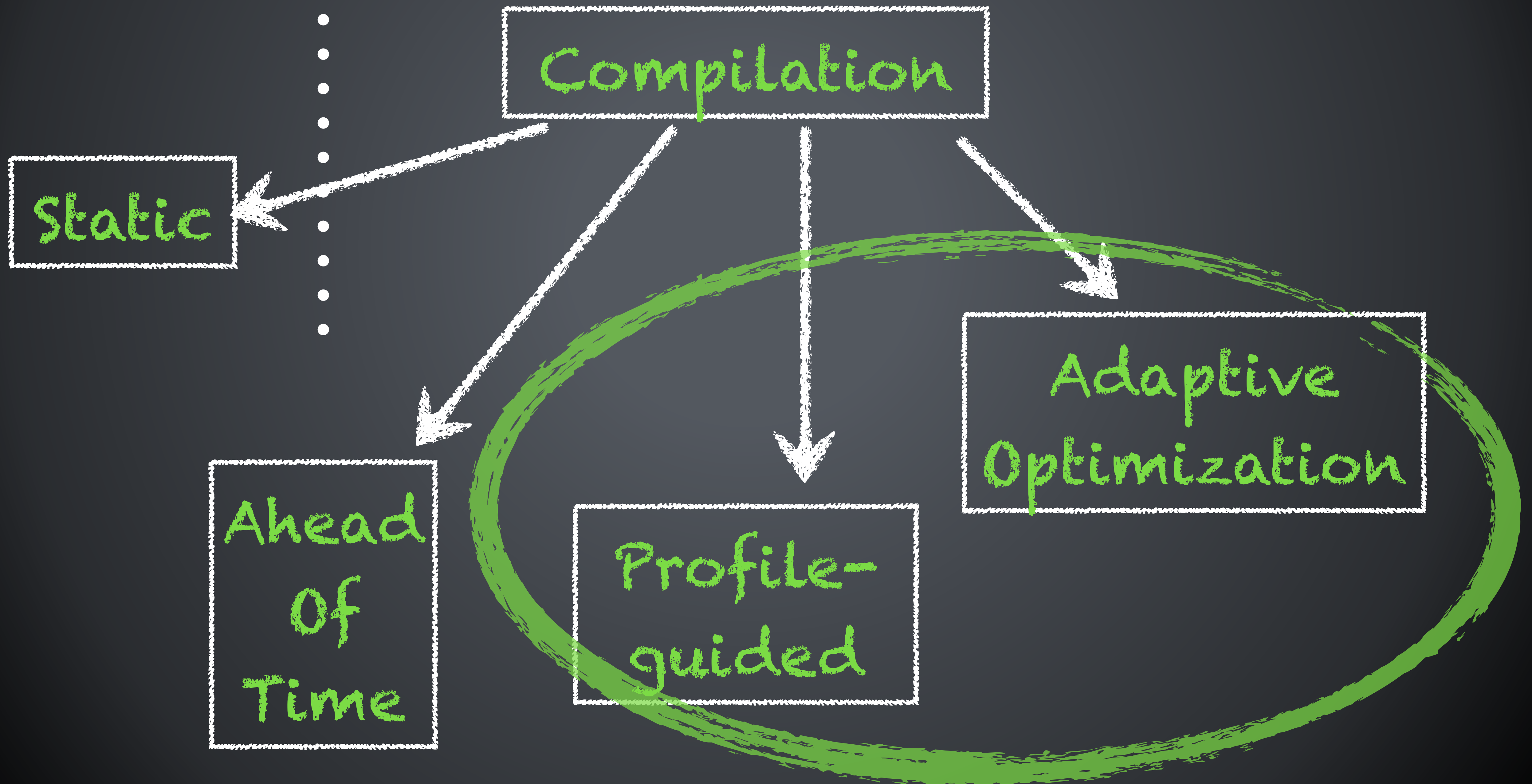
©2016 CodeKaram

Common Compilation Techniques



©2016 CodeKaram

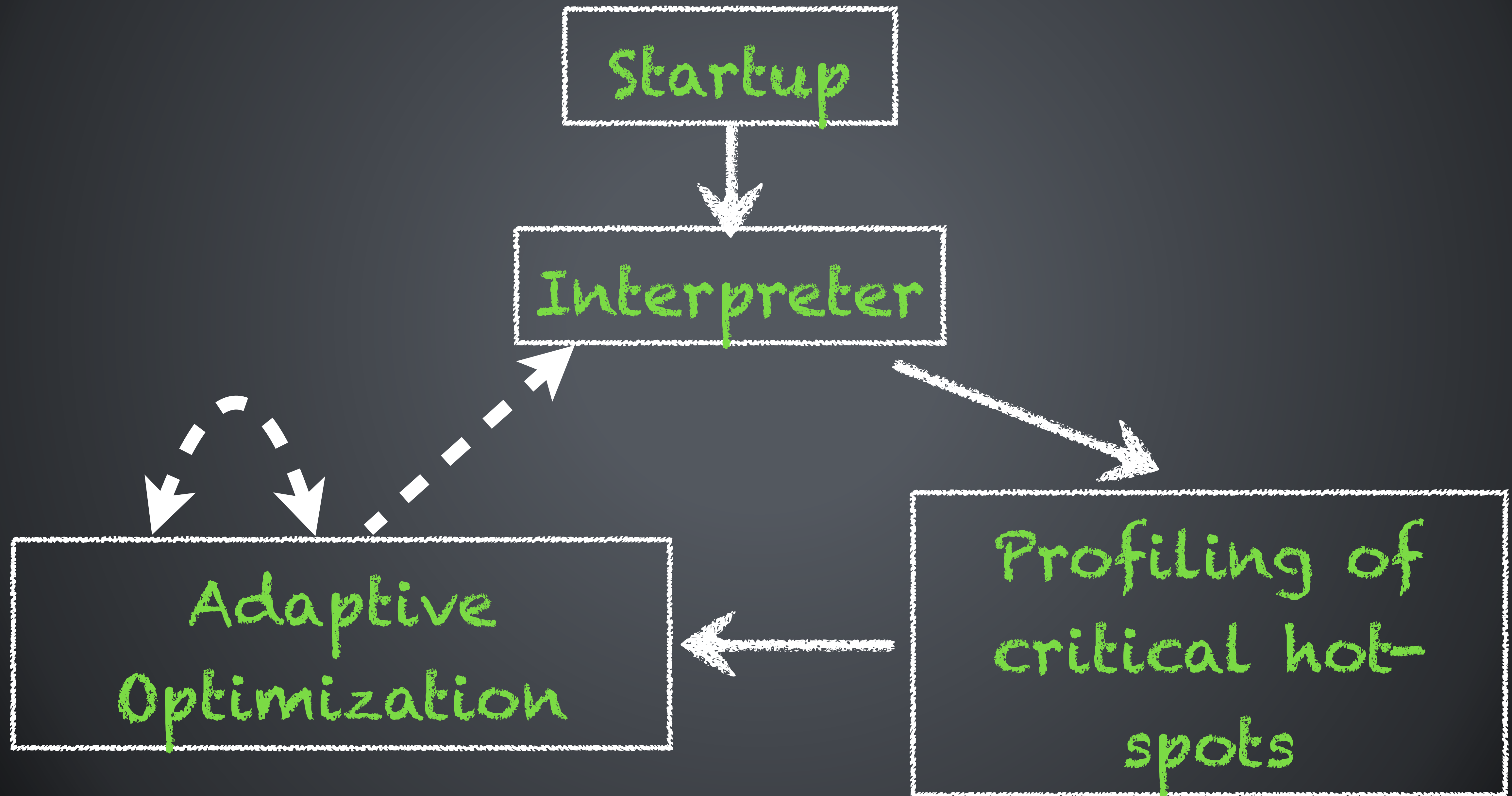
HotSpot VM



©2016 CodeKaram

Advanced JIT & Runtime Optimizations - Adaptive Optimization

The Plot



©2016 CodeKaram

Identifying Performance Critical Methods

- CompileThreshold
  - Identify root of compilation
  - Method Compilation or On-stack replacement (Loop)?

©2016 CodeKaram

Adaptive Optimization - Client Compiler (C1)

Client Compiler

- Start in interpreter
- CompileThreshold = 1500
  - Compile with client compiler
  - Few optimizations

©2016 CodeKaram

Adaptive Optimization - Server Compiler (C2)

Server Compiler

- Start in interpreter
- CompileThreshold = 10000
  - Compile with optimized server compiler
  - High performance optimizations

©2016 CodeKaram

Adaptive Optimization - Tiered Compilation

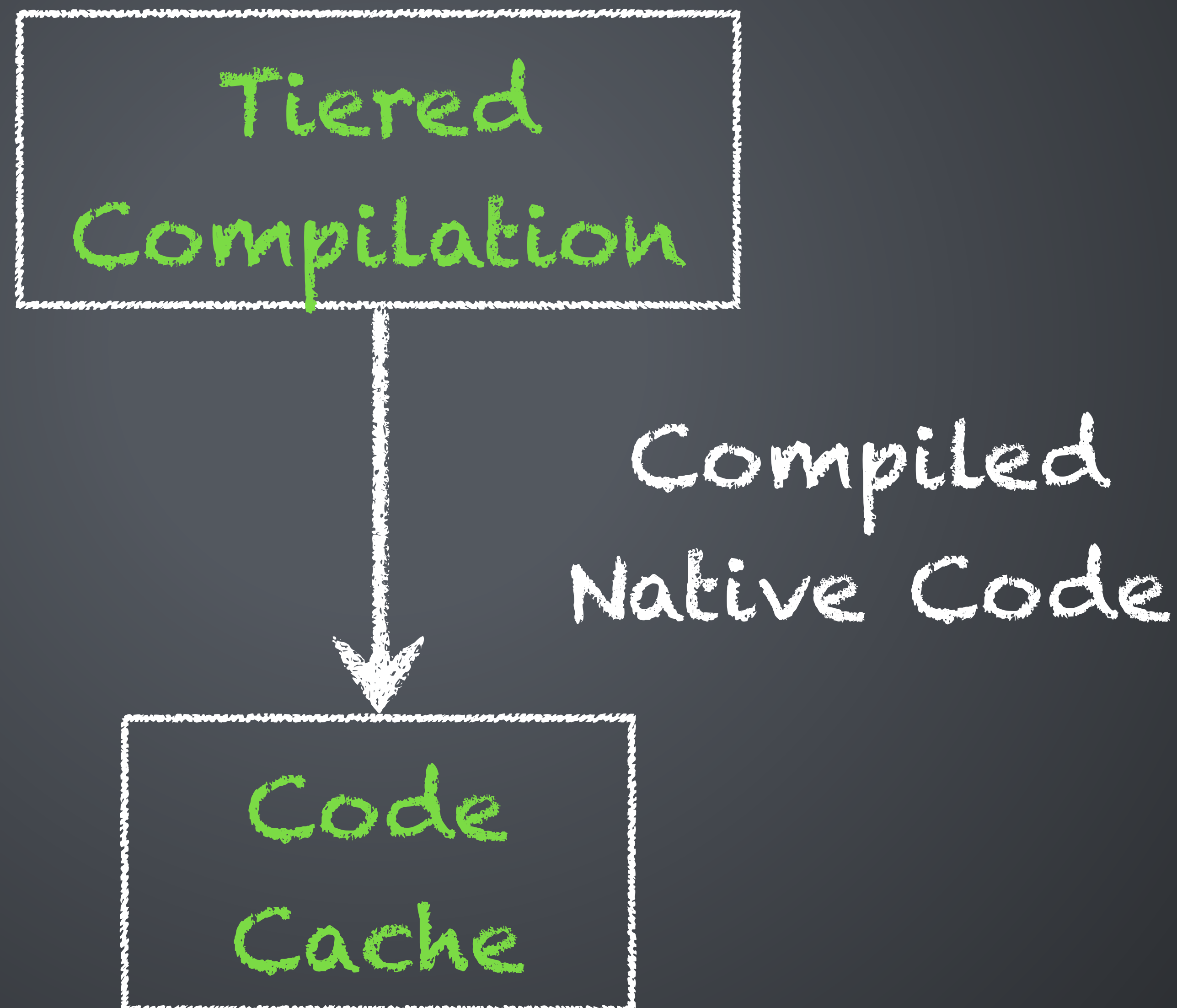
Just opening up a whole new
world of possibilities for you

Tiered Compilation

- Start in interpreter
- Tiered optimization with client compiler
  - Code profiled information
- Enable server compiler

©2016 CodeKaram

Tiered Compilation - Effect on Code Cache



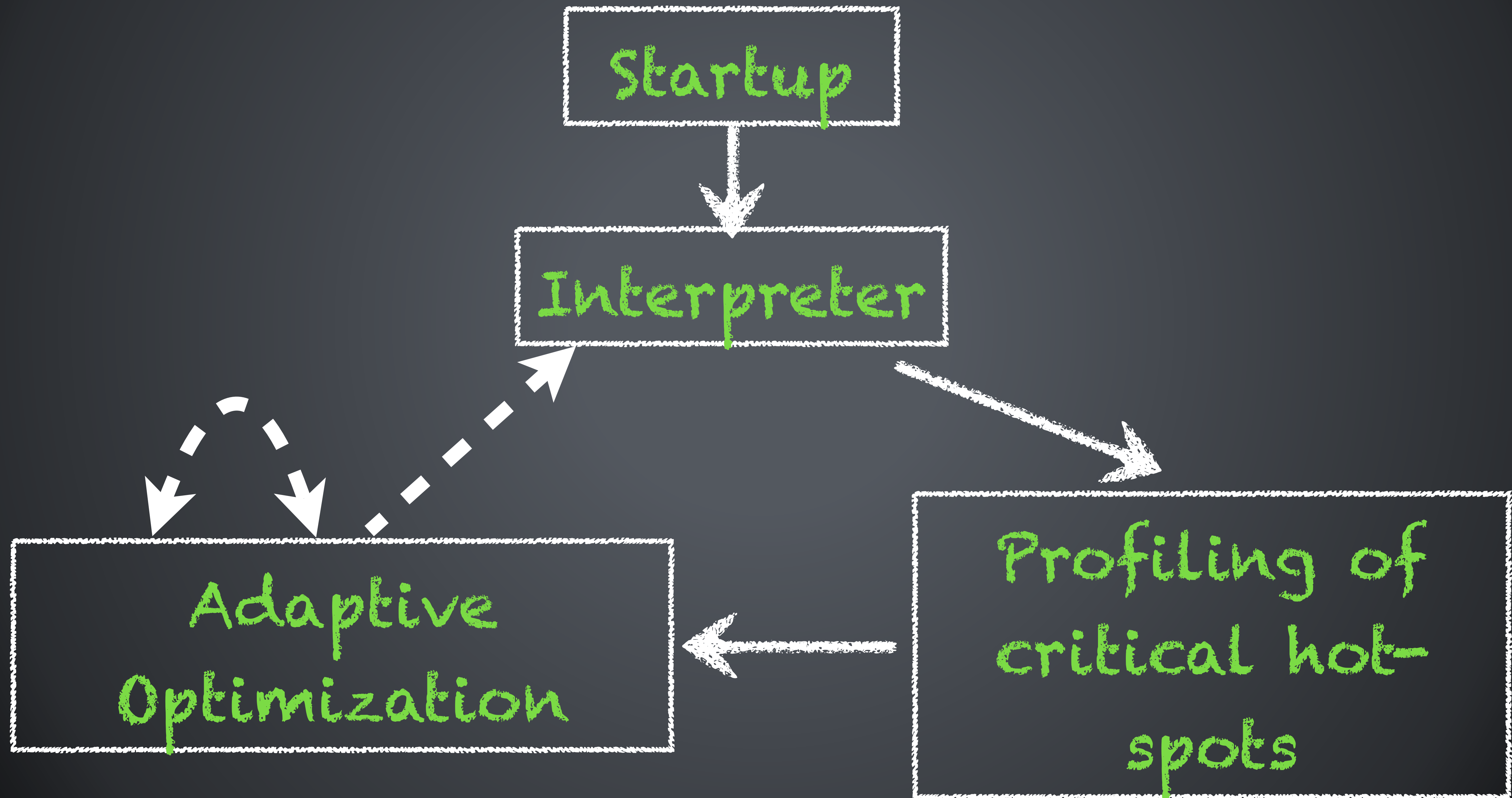
©2016 CodeKaram

Tiered Compilation - Effect on Code Cache

- C1 compilation threshold for tiered is about a 100 invocations
- Tiered Compilation has a lot more profiled information for C1 compiled methods
- CodeCache needs to be 5x larger than non-tiered
  - Default on JDK8 when tiered is enabled (240MB vs 48MB)
  - Need more? Use -XX:ReservedCodeCacheSize

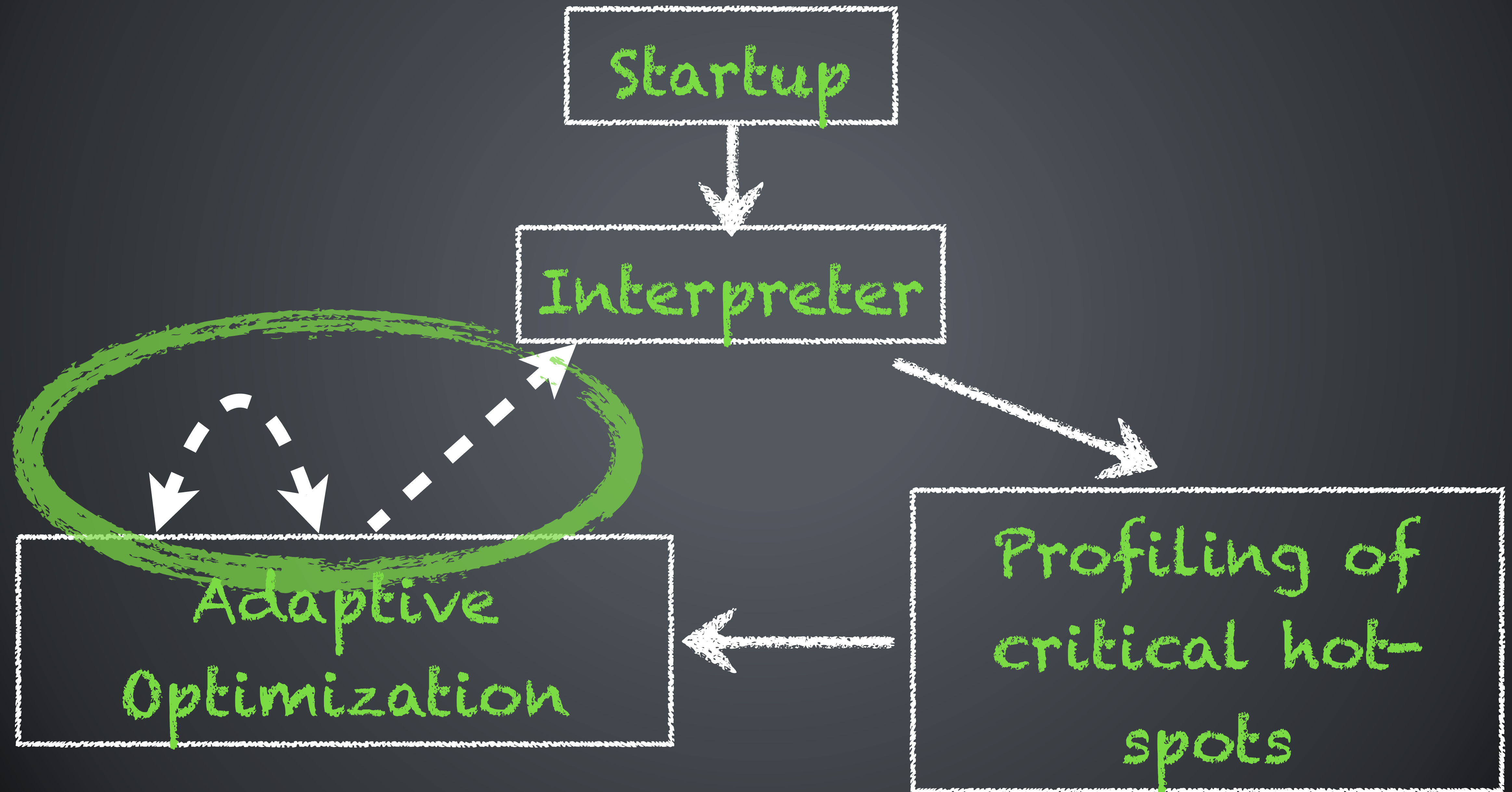
©2016 CodeKaram

The Plot



©2016 CodeKaram

The Plot



©2016 CodeKaram

Adaptive Optimization - Dynamic Deoptimization

Dynamic Deoptimization

©2016 CodeKaram

Dynamic Deoptimization

- Oopsie!
    - dependencies invalidation
    - classes unloading and redefinition
    - uncommon path in compiled code
    - misguided profiled information

©2016 CodeKaram

Understanding Adaptive Optimization - PrintCompilation

PrintCompilation

`timestamp compilation-id flags tiered-compilation-level Method <@ osr_bci> code-size <deoptimization>`

©2016 CodeKaram

PrintCompilation

```
Flags:
%: is_osr_method
s: is_synchronized
!: has_exception_handler
b: is_blocking
n: is_native
```


©2016 CodeKaram

PrintCompilation

```
    567  693 % !   3
org.h2.command.dml.Insert::insertRows @ 76 (513 bytes)

    656  797      n 0
java.lang.Object::clone (native)

    779  835  s    4
java.lang.StringBuffer::append (13 bytes)
```


©2016 CodeKaram

Dynamic Deoptimization

```
    573  704       2       org.h2.table.Table::fireAfterRow (17 bytes)
   7963 2223       4       org.h2.table.Table::fireAfterRow (17 bytes)
   7964  704       2       org.h2.table.Table::fireAfterRow (17 bytes) made not entrant
  33547  704       2       org.h2.table.Table::fireAfterRow (17 bytes) made zombie
```


©2016 CodeKaram

Dynamic Deoptimization

```
    573  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
   7963 2223       4
org.h2.table.Table::fireAfterRow (17 bytes)
   7964  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
made not entrant
  33547  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
made zombie
```


©2016 CodeKaram

Dynamic Deoptimization

```
    573  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
   7963 2223       4
org.h2.table.Table::fireAfterRow (17 bytes)
   7964  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
made not entrant
  33547  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
made zombie
```


©2016 CodeKaram

Dynamic Deoptimization

```
     573  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
    7963 2223       4
org.h2.table.Table::fireAfterRow (17 bytes)
    7964  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
made not entrant
   33547  704       2
org.h2.table.Table::fireAfterRow (17 bytes)
made zombie
```


©2016 CodeKaram

Dynamic Deoptimization

```
    573  704        2
org.h2.table.Table::fireAfterRow (17 bytes)
   7963 2223        4
org.h2.table.Table::fireAfterRow (17 bytes)
   7964  704        2
org.h2.table.Table::fireAfterRow (17 bytes)
made not entrant
  33547  704        2
org.h2.table.Table::fireAfterRow (17 bytes)
made zombie
```


©2016 CodeKaram

Adaptive Optimization - Inlining

Inlining

HotSpot has studied Inlining to it’s max potential!

©2016 CodeKaram

Inlining

MinInliningThreshold, MaxFreqInlineSize, InlineSmallCode, MaxInlineSize, MaxInlineLevel, DesiredMethodLimit …

©2016 CodeKaram

Understanding Adaptive Optimization - PrintInlining

PrintInling*

```
@ 76   java.util.zip.Inflater::setInput (74 bytes)
too big
@ 80   java.io.BufferedInputStream::getBufIfOpen (21
bytes)   inline (hot)
@ 91   java.lang.System::arraycopy (0 bytes)
(intrinsic)
@ 2   java.lang.ClassLoader::checkName (43 bytes)
callee is too large
```

* needs -XX:+UnlockDiagnosticVMOptions

©2016 CodeKaram

PrintInling*

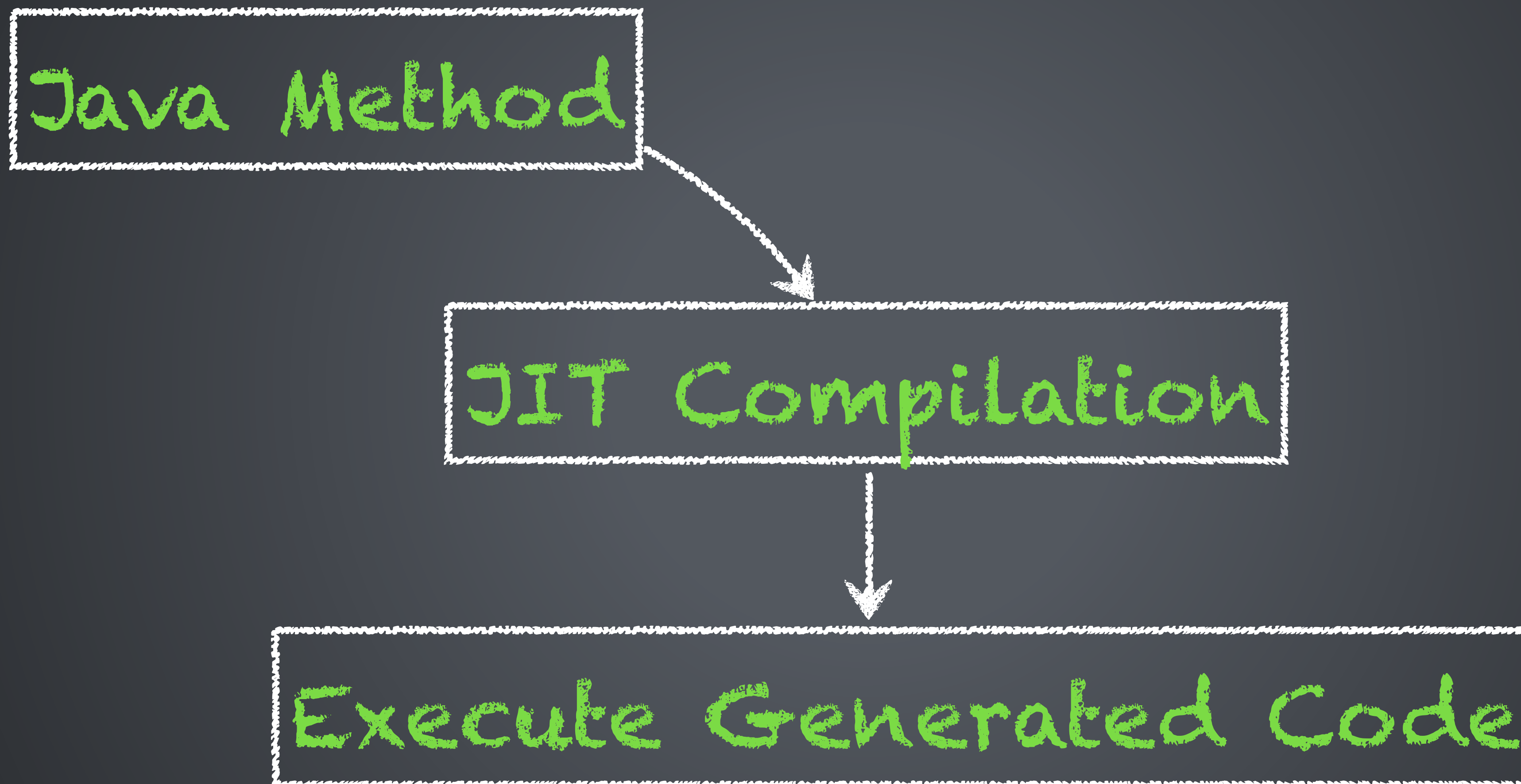
```
@ 76   java.util.zip.Inflater::setInput (74 bytes)
too big
@ 80   java.io.BufferedInputStream::getBufIfOpen (21
bytes)   inline (hot)
@ 91   java.lang.System::arraycopy (0 bytes)
(intrinsic)
@ 2   java.lang.ClassLoader::checkName (43 bytes)
callee is too large
```

* needs -XX:+UnlockDiagnosticVMOptions

©2016 CodeKaram

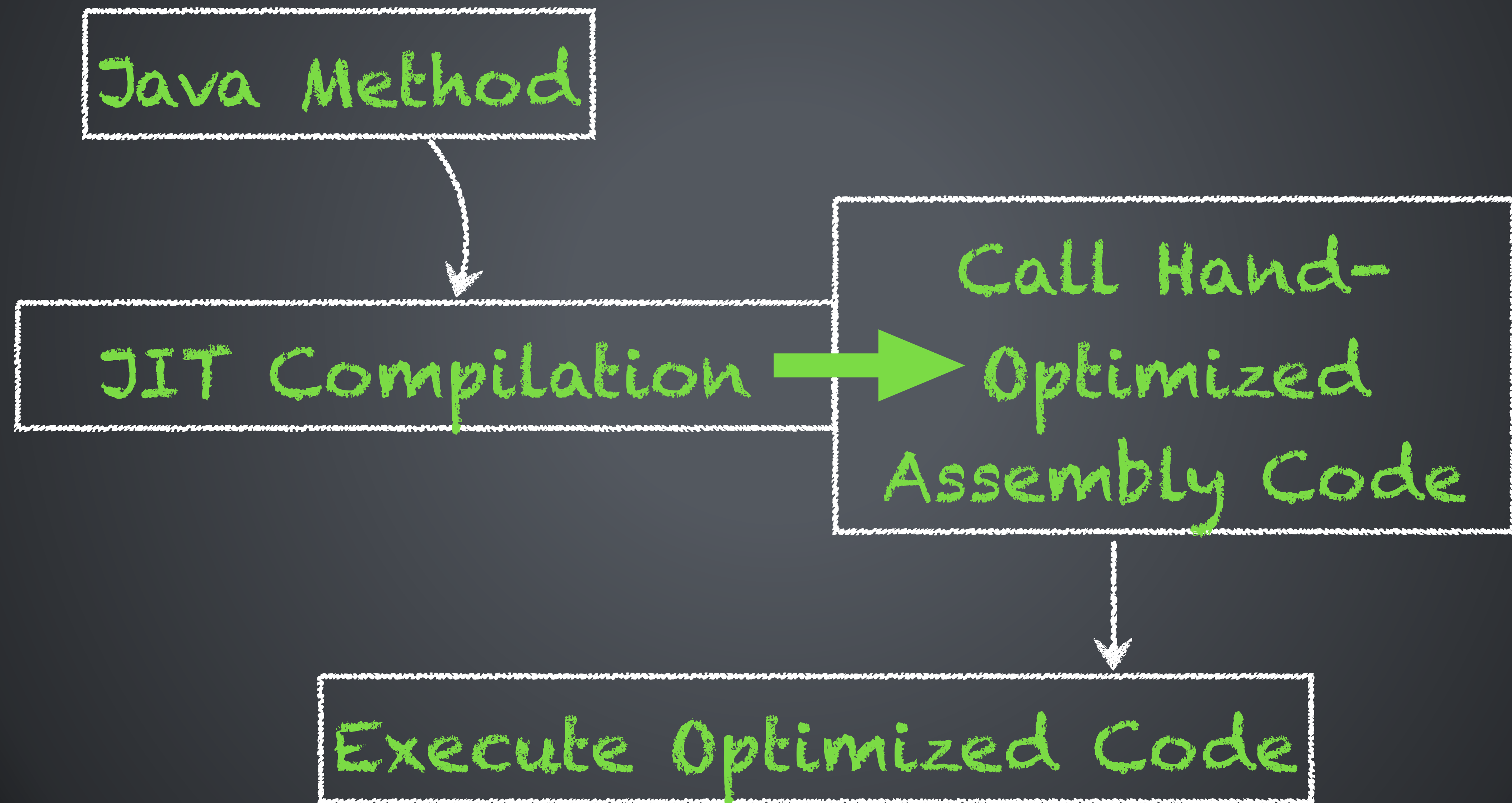
Adaptive Optimization - Intrinsics

Without Intrinsics



©2016 CodeKaram

Intrinsics



©2016 CodeKaram

PrintInling*

```
@ 76   java.util.zip.Inflater::setInput (74 bytes)
too big
@ 80   java.io.BufferedInputStream::getBufIfOpen (21
bytes)   inline (hot)
@ 91   java.lang.System::arraycopy (0 bytes)
(intrinsic)
@ 2   java.lang.ClassLoader::checkName (43 bytes)
callee is too large
```

* needs -XX:+UnlockDiagnosticVMOptions

©2016 CodeKaram

Adaptive Optimization - Vectorization

Vectorization in HotSpot VM??

Vectorization

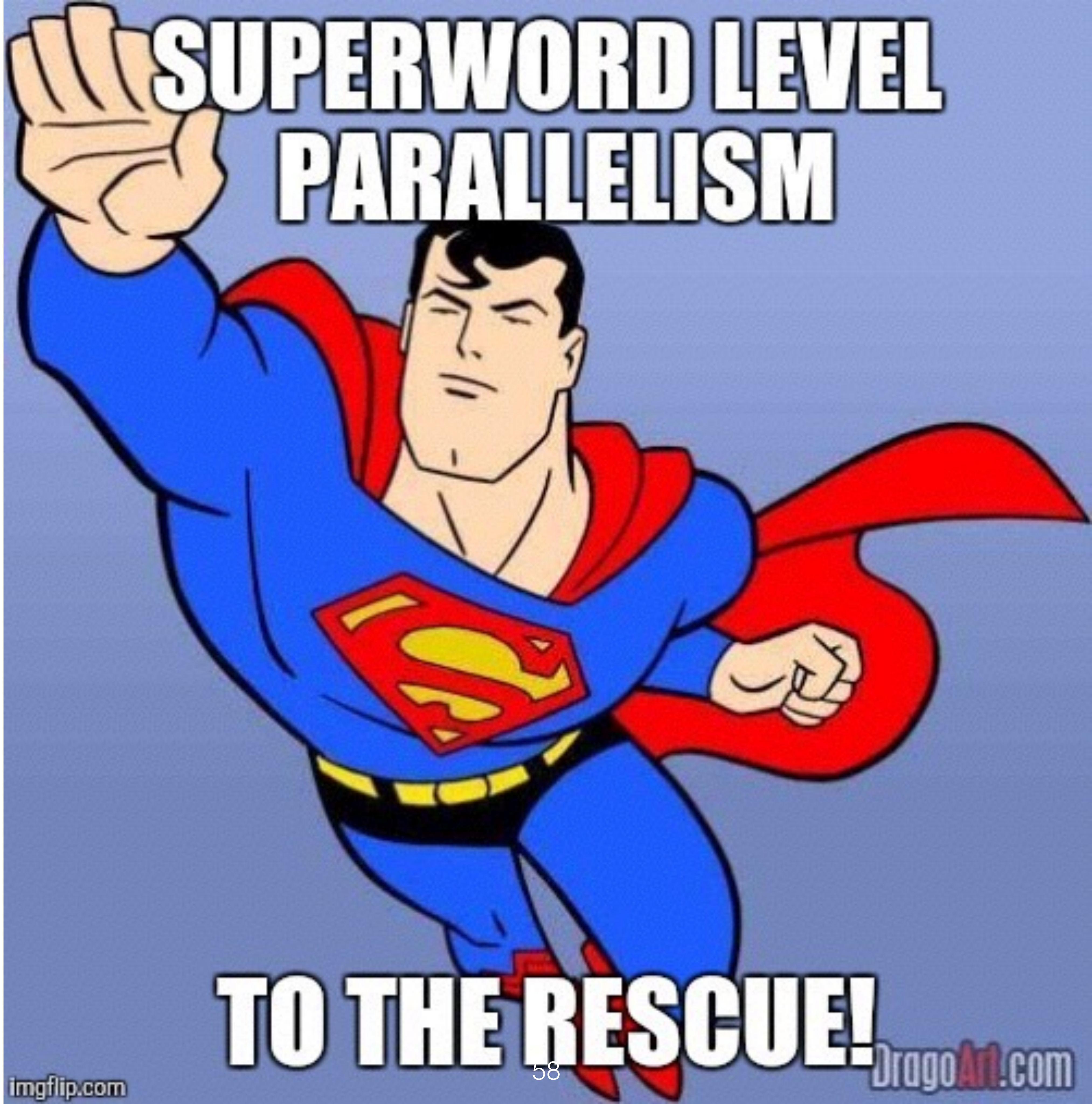
- Utilize SIMD (Single Instruction Multiple Data) instructions offered by the processor.
  - Generate assembly stubs
- Get benefits of operating on cache line size data chunks

©2016 CodeKaram

Adaptive Optimization - Auto-Vectorization

Auto-Vectorization in HotSpot VM??

Look Ma, no stubs!!


SUPERWORD LEVEL PARALLELISM
TO THE RESCUE!
DragoArt.com
imgflip.com

SuperWord Level Parallelism

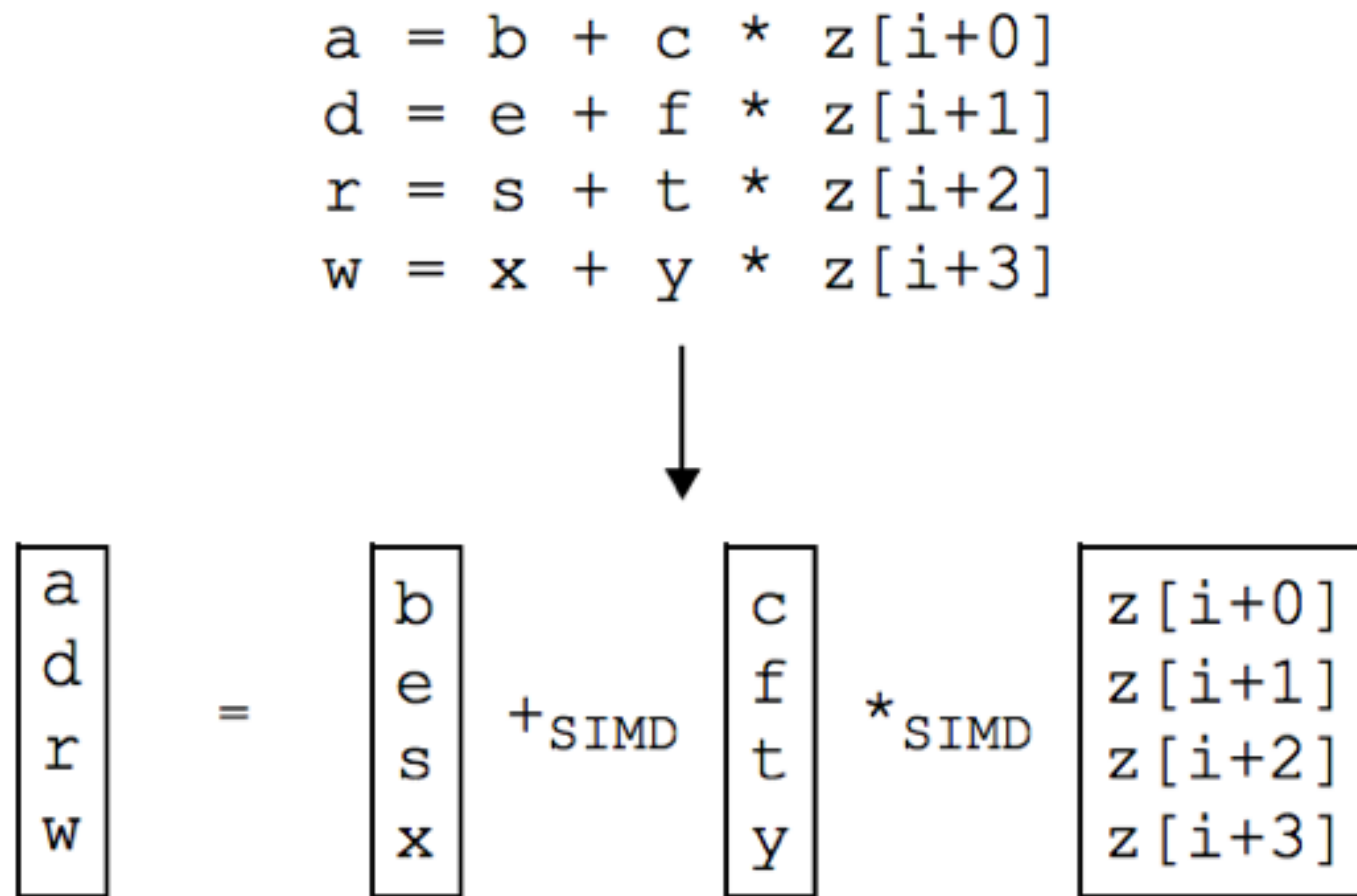


Figure 1: Isomorphic statements that can be packed and executed in parallel.

http://groups.csail.mit.edu/cag/slp/SLP-PLDI-2000.pdf

©2016 CodeKaram

SuperWord Level Parallelism

- Loop Unrolling
- Alignment Analysis
- Pre-Optimization
- Identifying Adjacent Memory References
- Extending the “PackSet”
- Combination and SIMD operation

http://groups.csail.mit.edu/cag/slp/SLP-PLDI-2000.pdf

©2016 CodeKaram

Other Adaptive Optimization

Other JIT Compiler Optimizations

- Fast dynamic type tests for type safety
- Range check elimination
- Loop unrolling
- Escape Analysis

©2016 CodeKaram

Other JIT Compiler Optimizations

- Fast dynamic type tests for type safety
- Range check elimination
- Loop unrolling
- Escape Analysis

©2016 CodeKaram

Adaptive Optimization - Escape Analysis

Escape Analysis

- Entire IR graph
- escaping allocations?
  - not stored to a static field or non-static field of an external object,
  - not returned from method,
  - not passed as parameter to another method where it escapes.

©2016 CodeKaram

Escape Analysis

allocated object doesn't escape the compiled method

+

allocated object not passed as a parameter

=

remove allocation and keep field values in registers

©2016 CodeKaram

Escape Analysis

allocated object doesn't escape the compiled method

+

allocated object is passed as a parameter

=

remove locks associated with object and use optimized compare instructions

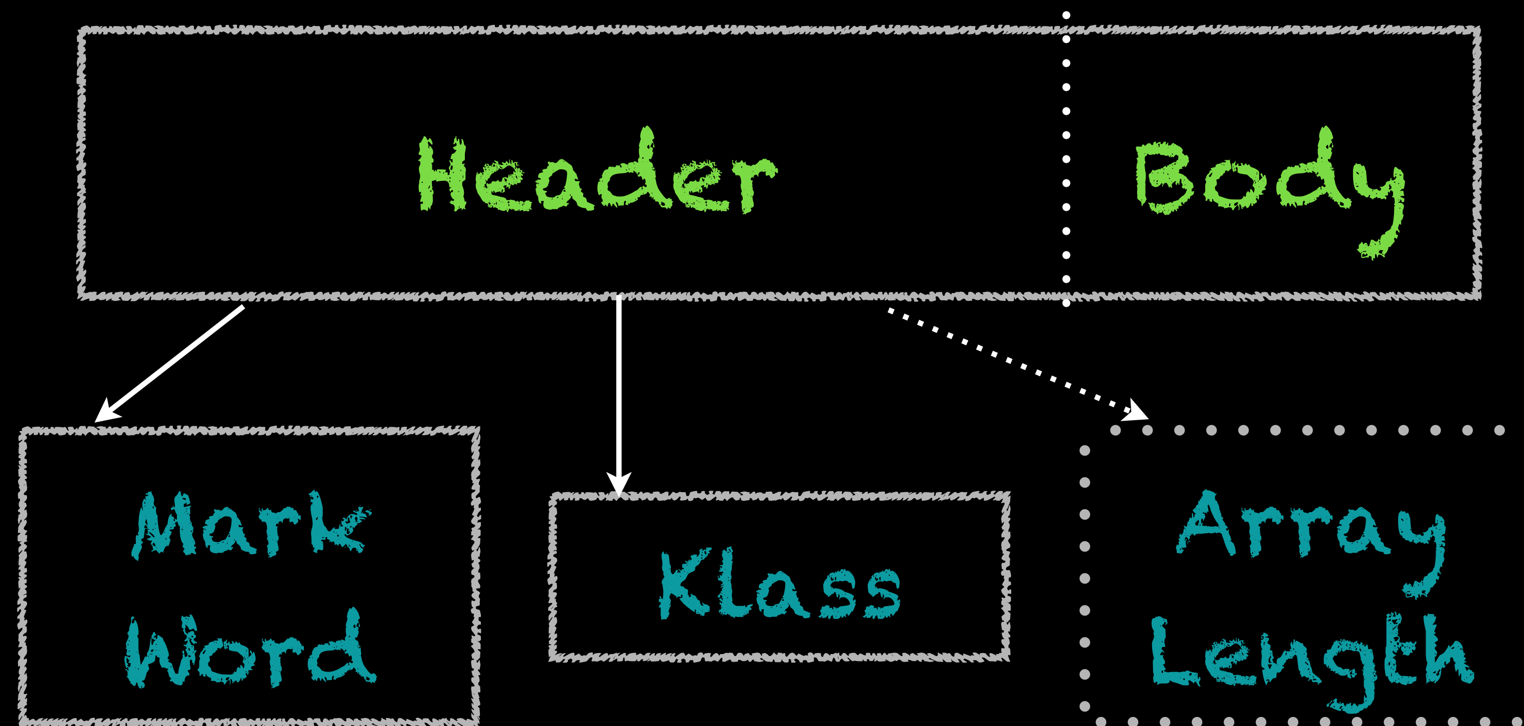
©2016 CodeKaram

Time Check!

©2016 CodeKaram

Advanced JIT & Runtime Optimizations - Compressed Oops

A Java Object



©2016 CodeKaram

Objects, Fields & Alignment

- Objects are 8 byte aligned (default).
- Fields:
    - are aligned by their type.
    - can fill a gap that maybe required for alignment.
    - are accessed using offset from the start of the object

©2016 CodeKaram

ILP32 vs. LP64 Field Sizes

Mark Word - 32 bit vs 64 bit

Klass - 32 bit vs 64 bit

Array Length - 32 bit on both

boolean, byte, char, float, int, short - 32 bit on both

double, long - 64 bit on both

©2016 CodeKaram

Compressed OOPs

<wide-oop> = <narrow-oop-base> + (<narrow-oop> << 3) + <field-offset>

©2016 CodeKaram

Compressed OOPs

Heap Size?	<4 GB (no encoding/ decoding needed)	>4GB; <28GB (zero-based)
<wide-oop> =	<narrow-oop>	<narrow-oop> << 3

©2016 CodeKaram

Compressed OOPs

Heap Size?	>28 GB; <32 GB (regular)	>32 GB; <64 GB * (change alignment)
<wide-oop> =	<narrow-oop-base> + (<narrow-oop> << 3) + <field-offset>	<narrow-oop-base> + (<narrow-oop> << 4) + <field-offset>

©2016 CodeKaram

Compressed OOPs

Heap Size?	<4 GB	>4GB; <28GB	<32GB	<64GB
Object Alignment?	8 bytes	8 bytes	8 bytes	16 bytes
Offset Required?	No	No	Yes	Yes
Shift by?	No shift	3	3	4

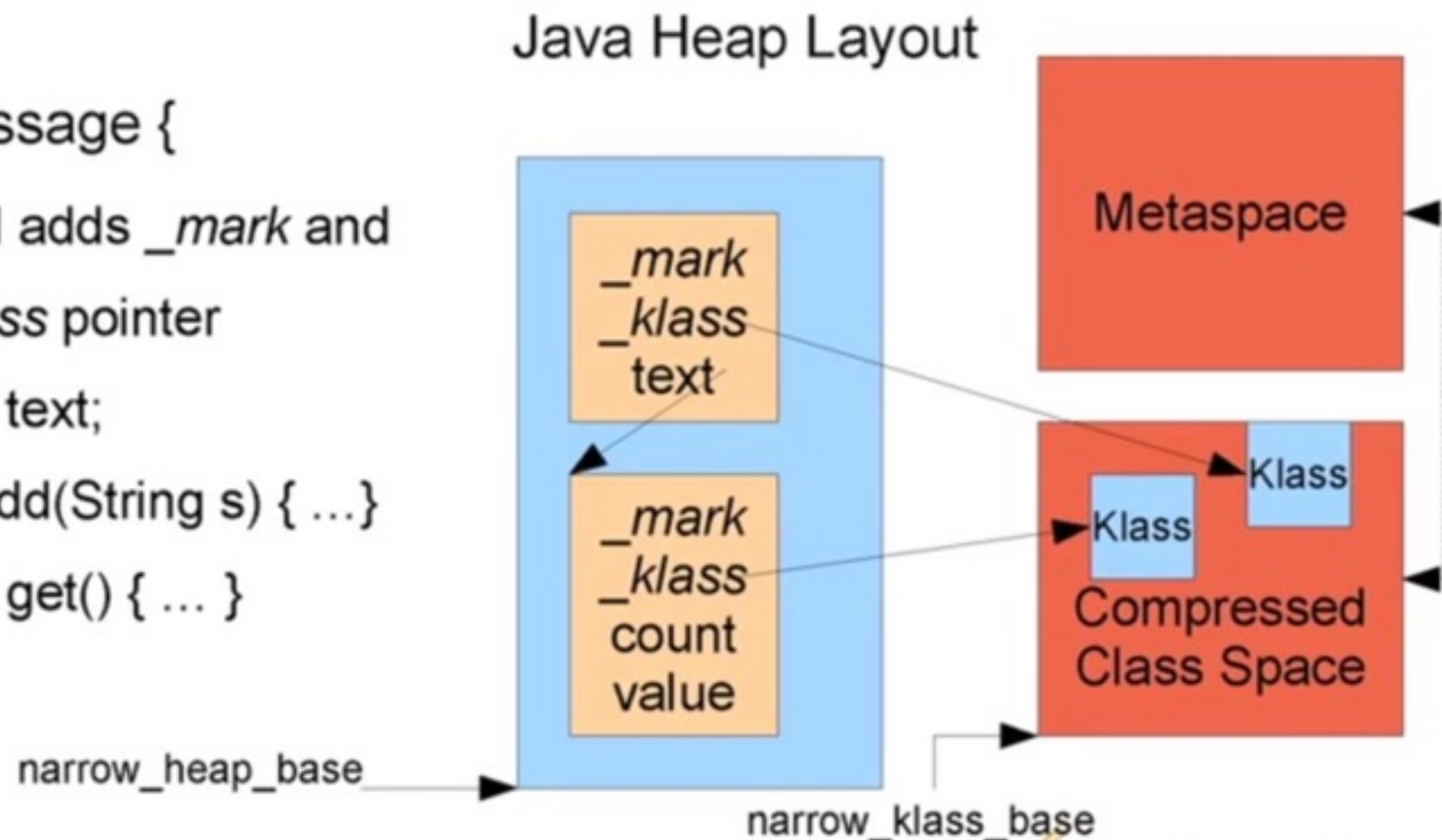
©2016 CodeKaram

Compressed Class Pointers

- JDK 8 —> Perm Gen Removal —> Class Data outside of heap
- Compressed class pointer space
  - contains class metadata
  - is a part of Metaspace

©2016 CodeKaram

Compressed Class Pointers



PermGen Removal Overview by Coleen Phillimore + Jon Masamitsu @JavaOne 2013

©2016 CodeKaram

Q & A

©2016 CodeKaram