JVM internals

Agenda

- Hotspot JVM
- Graal VM
- Garbage Collectors

Hotspot JVM

Virtual machines

- A typical virtual machine for an interpreted language provides:
  - Compilation of source language into VM specific bytecode
  - Data structures to contains instructions and operands (the data the instructions process)
  - A call stack for function call operations

Virtual machines

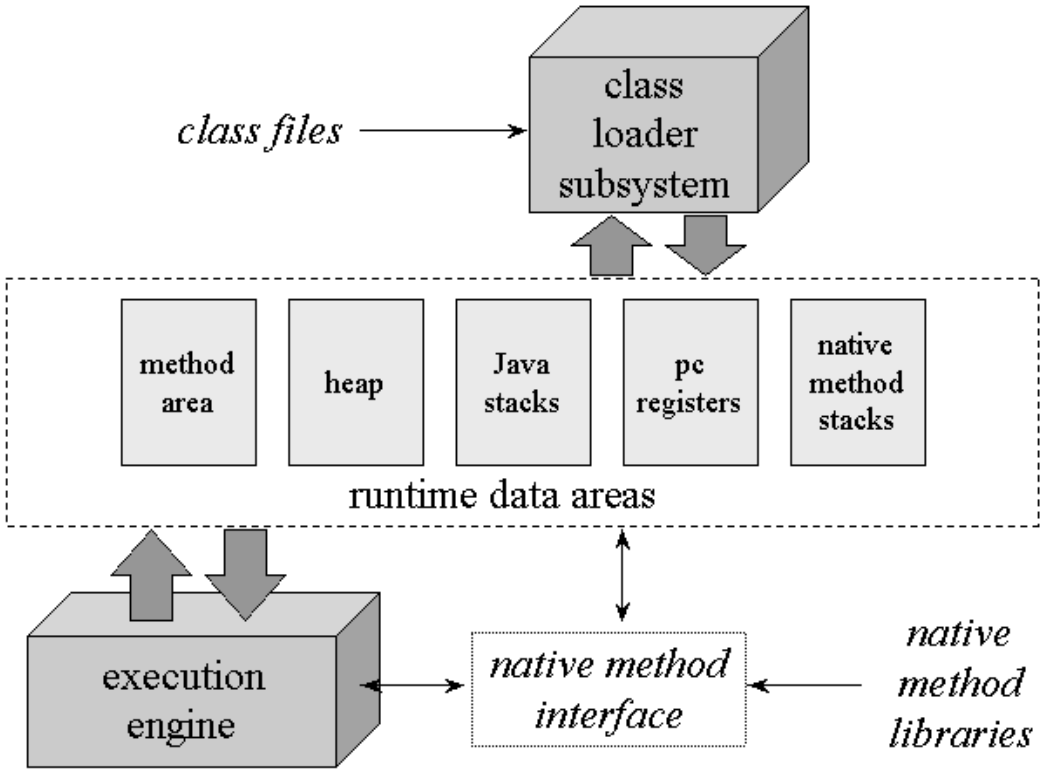
- A typical virtual machine for an interpreted language provides:
  - An ‘Instruction Pointer’ (IP) pointing to the next instruction to execute
  - A virtual ‘CPU’ – the instruction dispatcher that:
    - Fetches the next instruction (addressed by the instruction pointer)
    - Decodes the operands
    - Executes the instruction

The HotSpot JVM

- HotSpot is the standard JVM distribution of Oracle that provides:
  - bytecode execution - using an interpreter, two runtime compilers and On-Stack Replacement
  - storage allocation and garbage collection
  - runtimes - start up, shut down, class loading, threads, interaction with OS and others

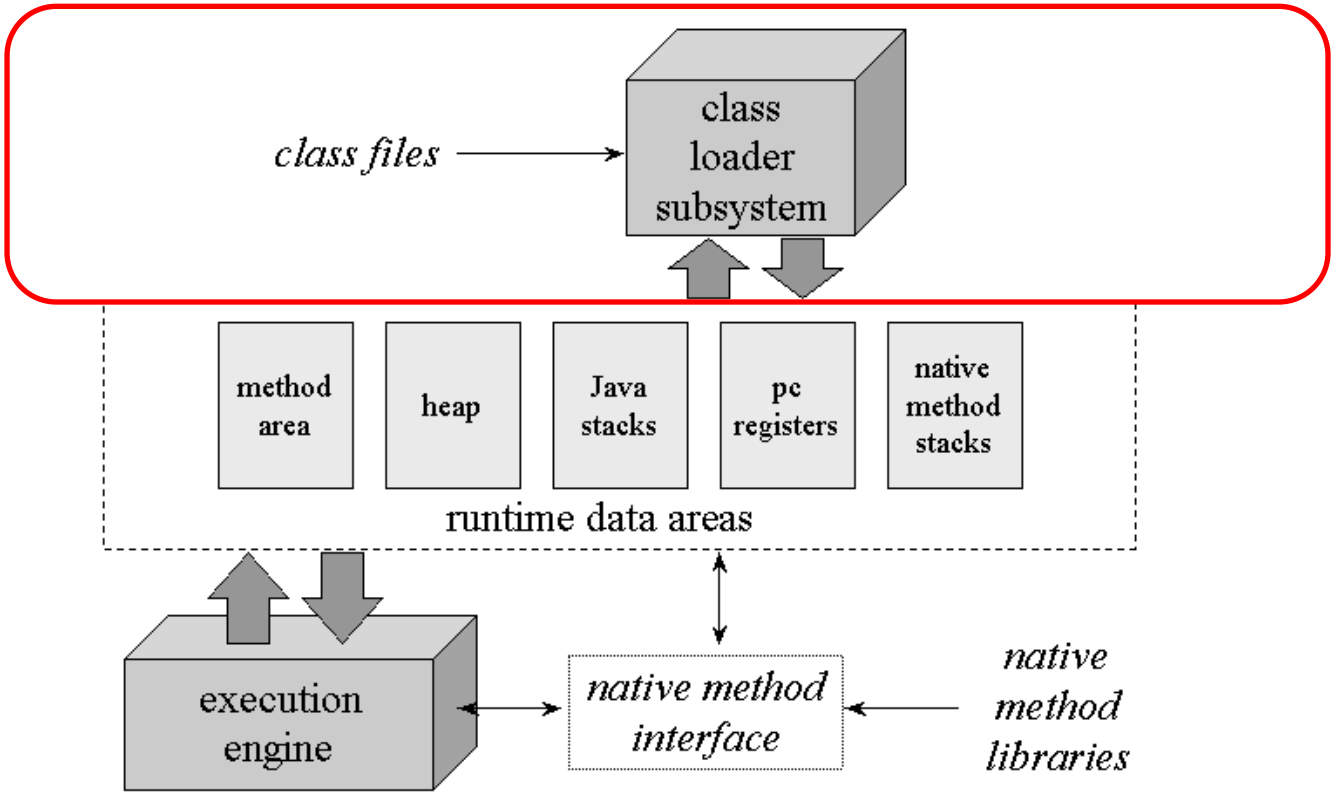
The HotSpot JVM

- Architecture:



The HotSpot JVM

- Architecture:



The HotSpot JVM

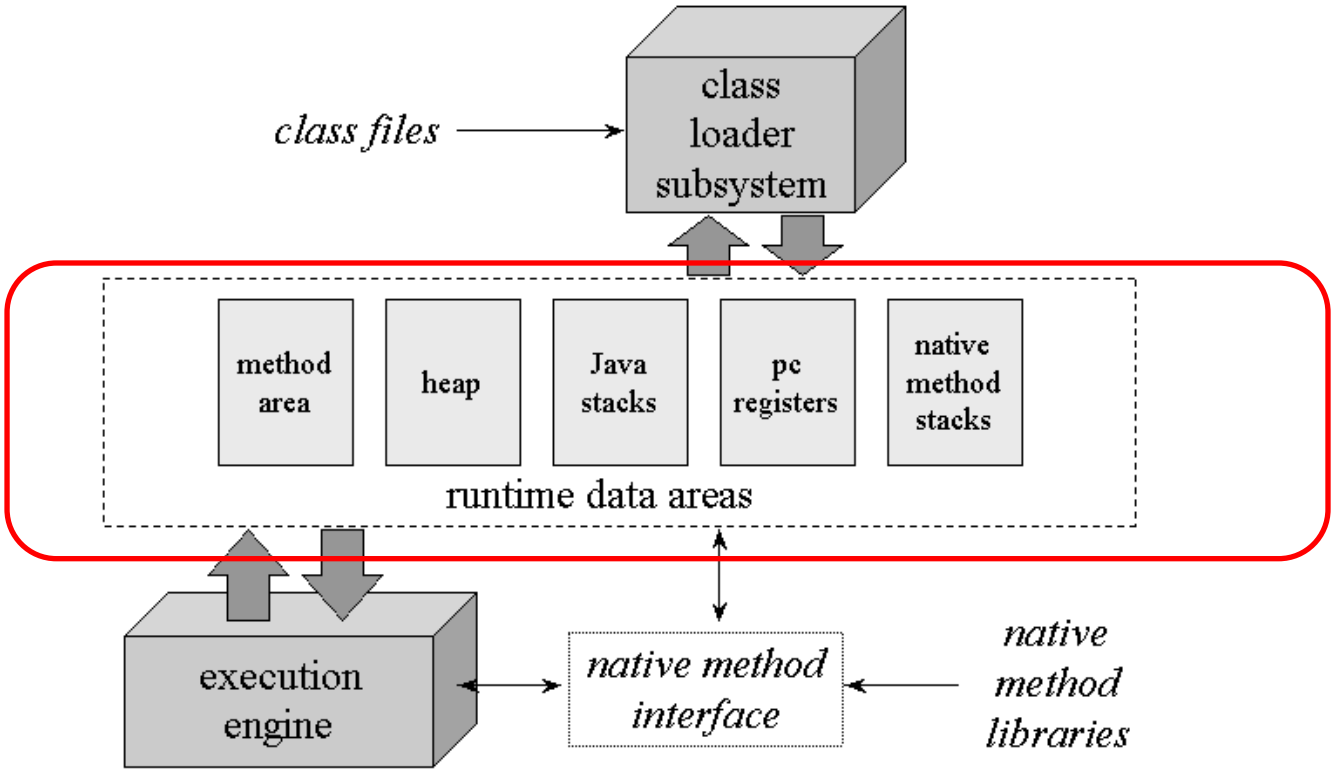
```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    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 HotSpot JVM

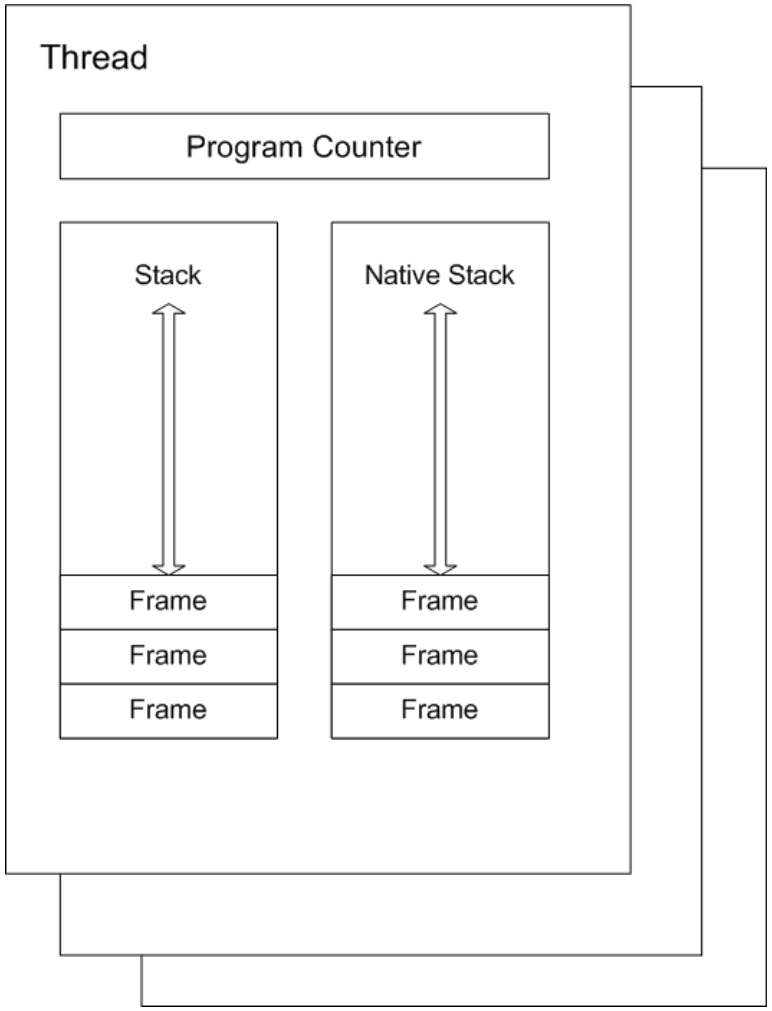
- Three phases of class-loading:
  - Loading
  - Linking
  - Initialization

The HotSpot JVM

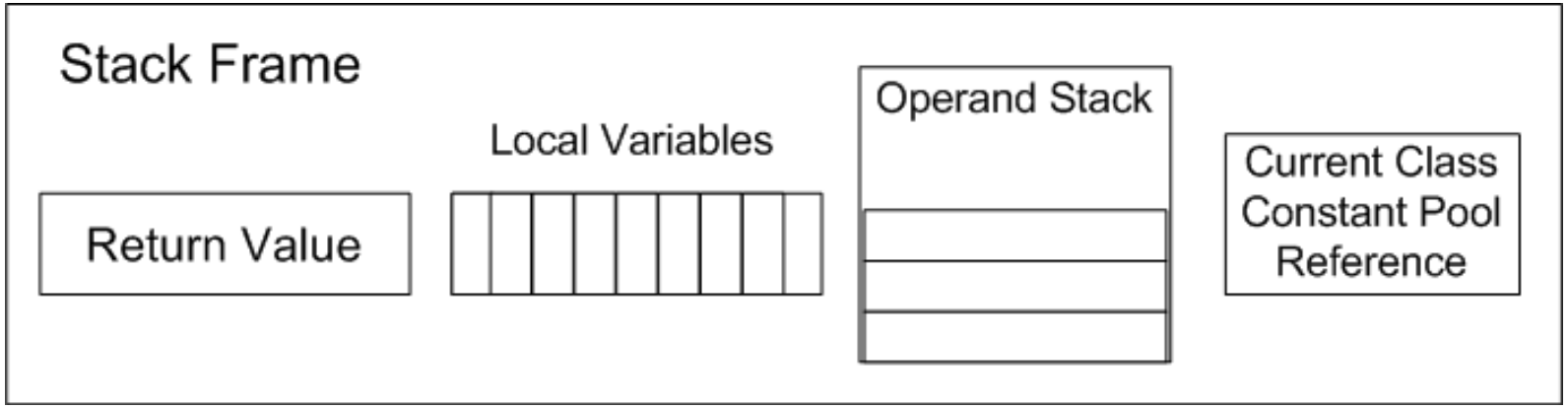
- Architecture:



The HotSpot JVM



The HotSpot JVM

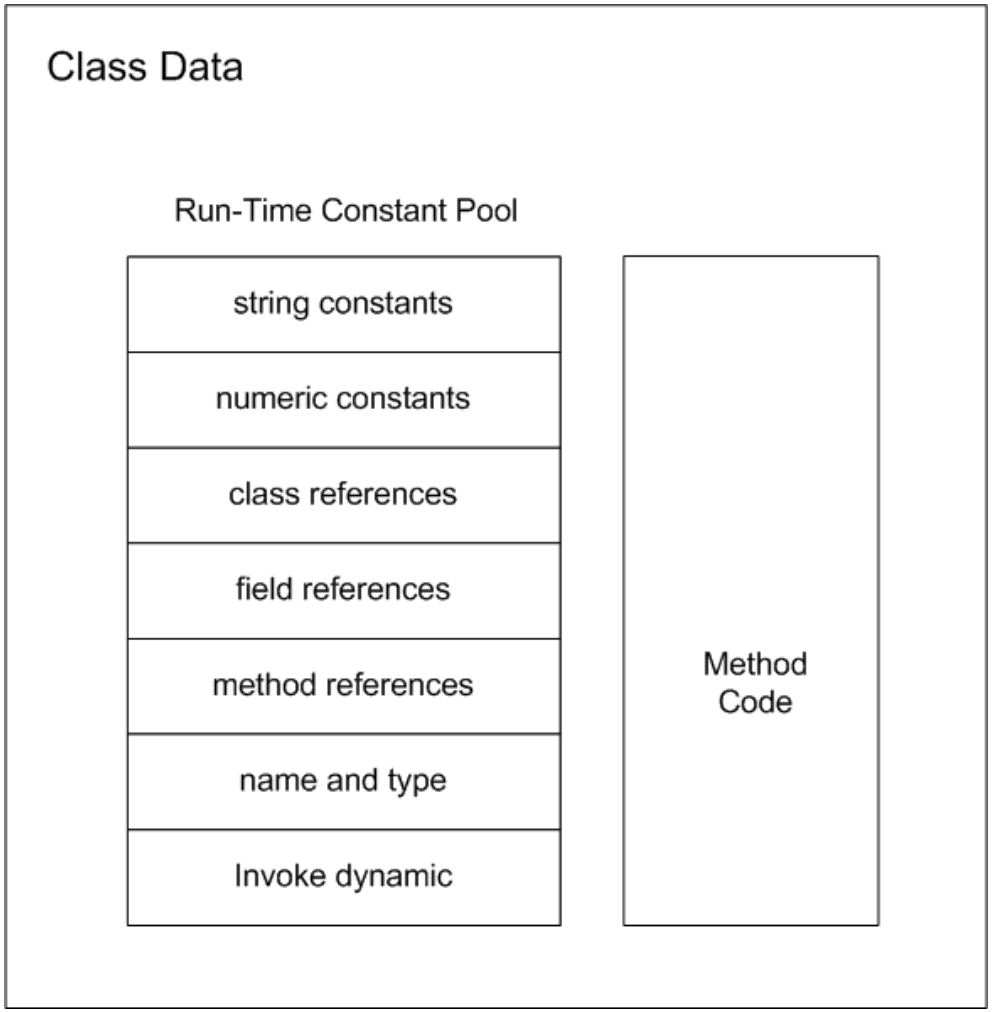


The HotSpot JVM

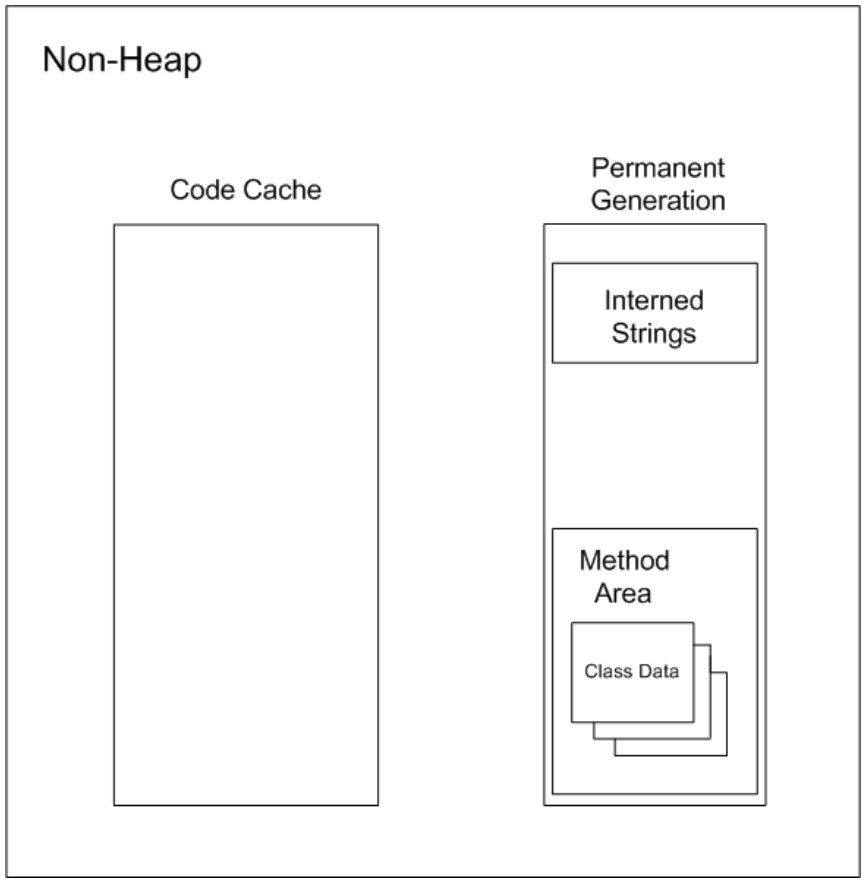
```
static int volume(int width,
        int depth,
        int height) {
    int area = width * depth;
    int volume = area * height;
    return volume;
}
```

```
0 iload_0
1 iload_1
2 imul
3 istore_3
4 iload_3
5 iload_2
6 imul
7 istore 4
9 iload 4
11ireturn
```

The HotSpot JVM

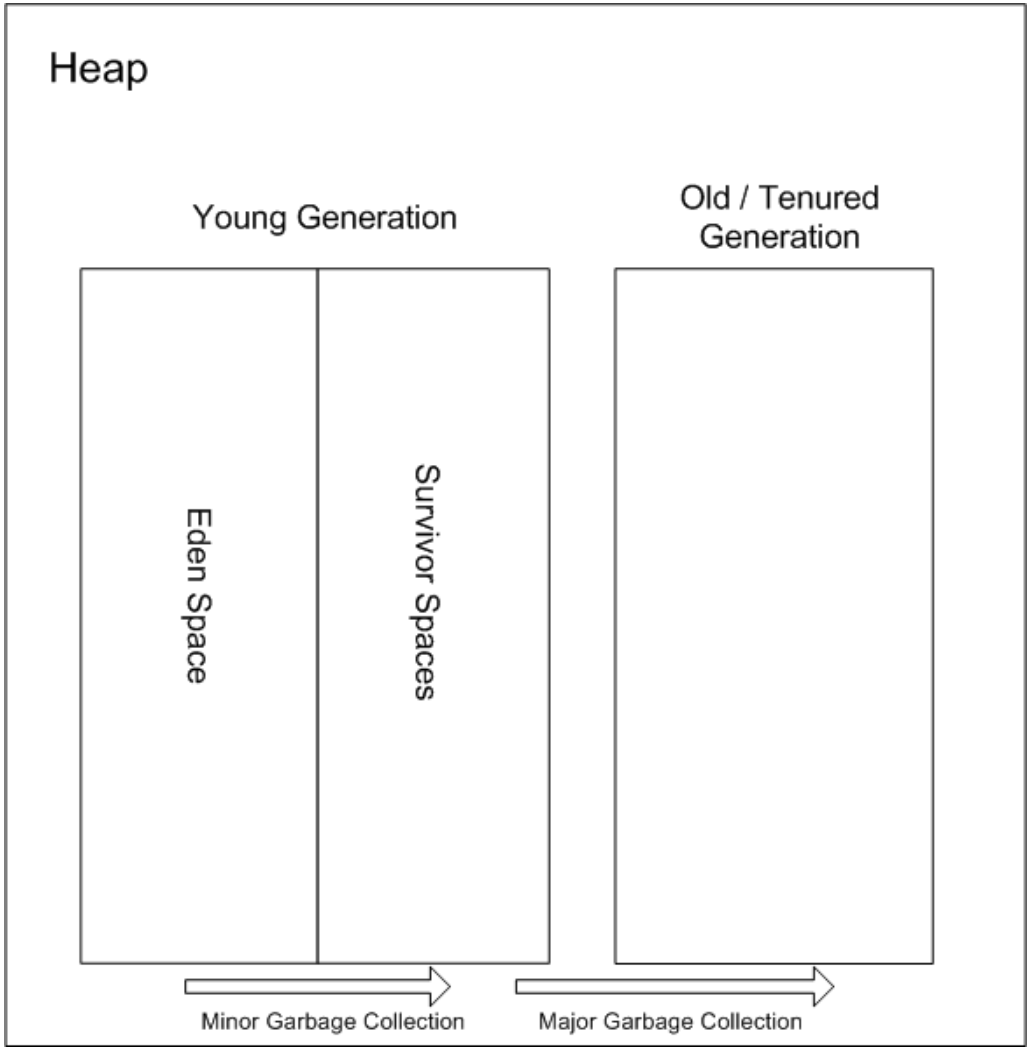


The HotSpot JVM



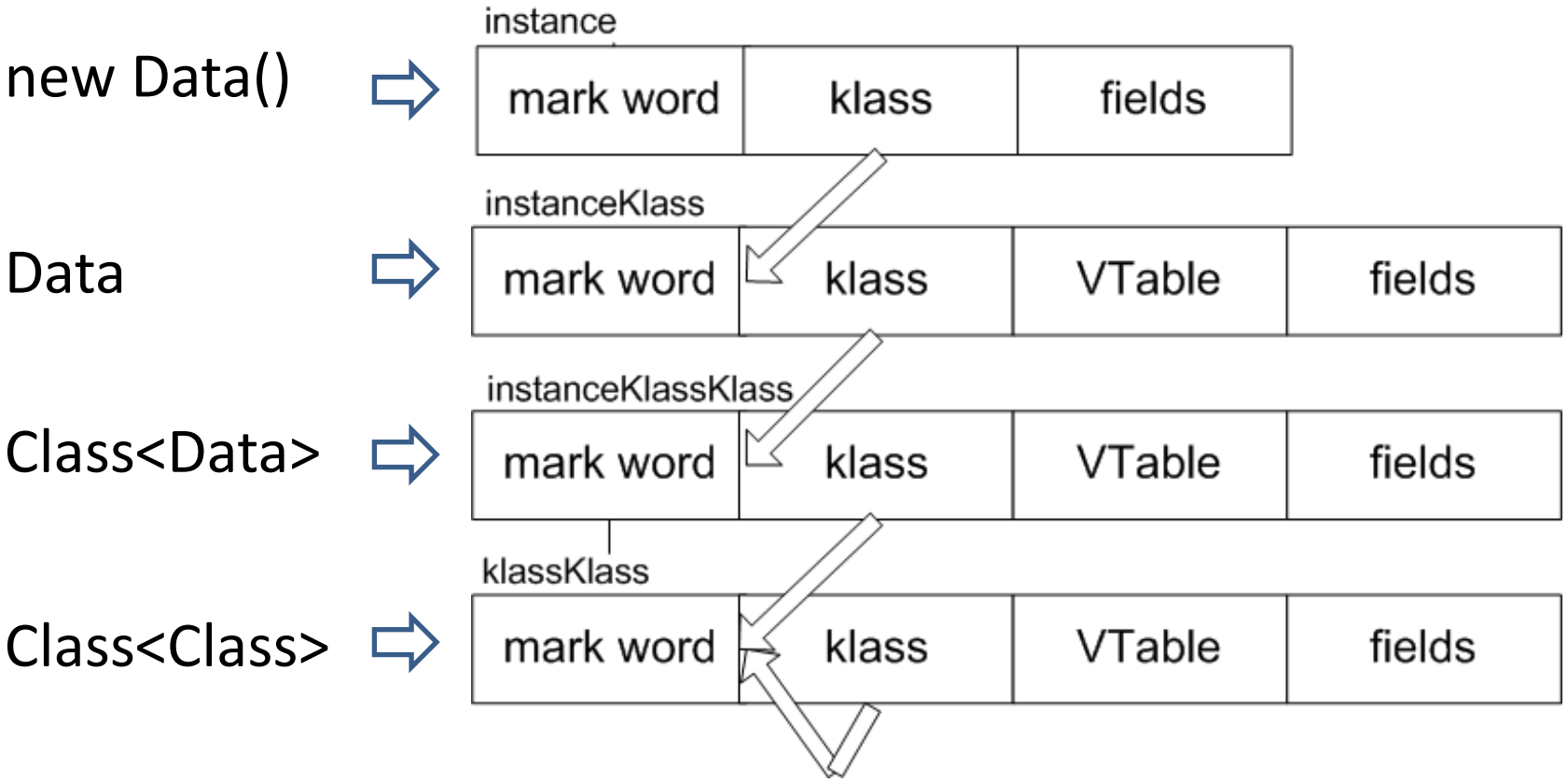
As of JDK 8 PermGen space is part of the heap

The HotSpot JVM



The HotSpot JVM

- Heap memory:

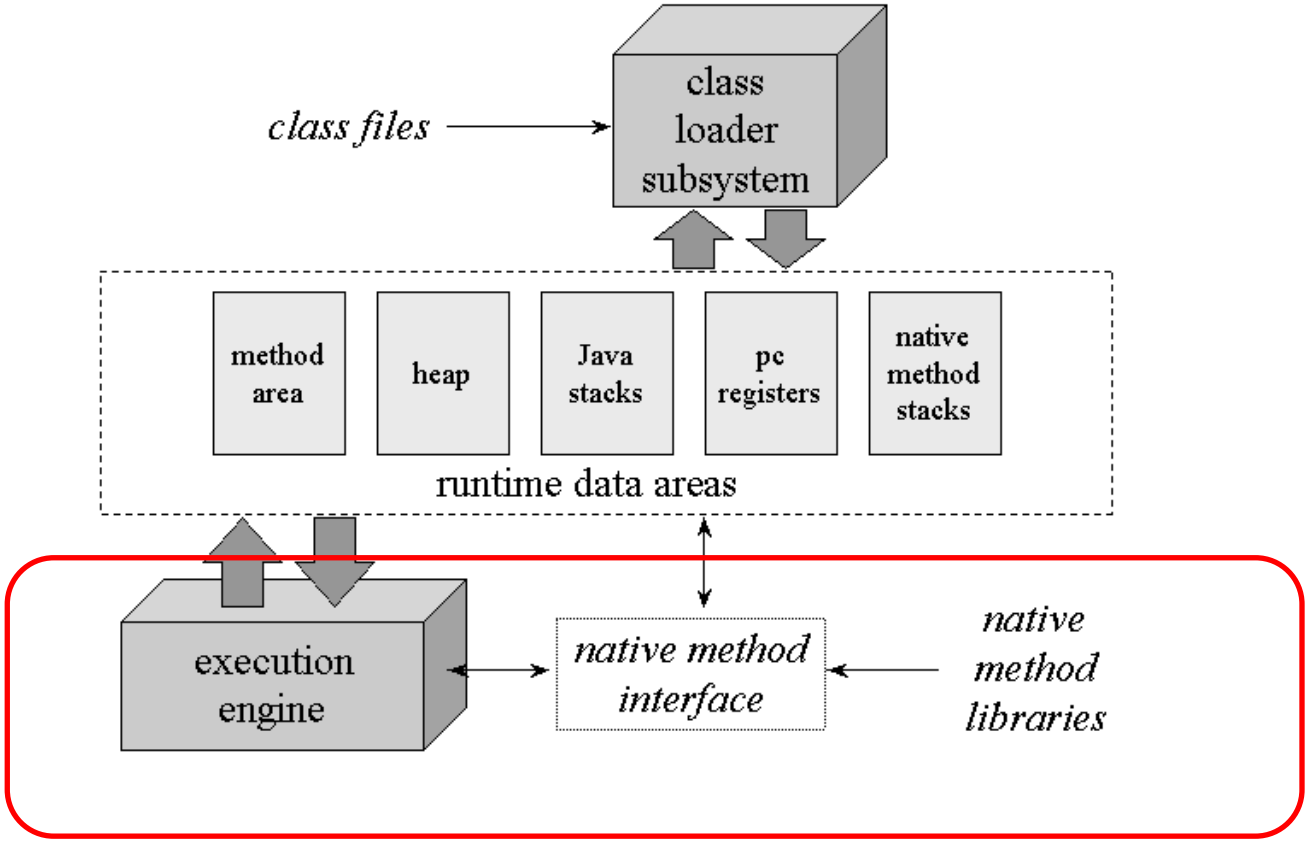


The HotSpot JVM

- Mark word contains:
  - identity hash code
  - age
  - lock record address
  - monitor address
  - state (unlocked, light-weight locked, heavy-weight locked, marked for GC)
  - biased / biasable (includes other fields such as thread ID)

The HotSpot JVM

- Architecture:



The HotSpot JVM

- Execution engine:

```
while(true) {
        bytecode b = bytecodeStream[pc++];
        switch(b) {
                case iconst_1: push(1); break;
                case iload_0: push(local(0)); break;
                case iadd: push(pop() + pop()); break;
        }
}
```

The HotSpot JVM

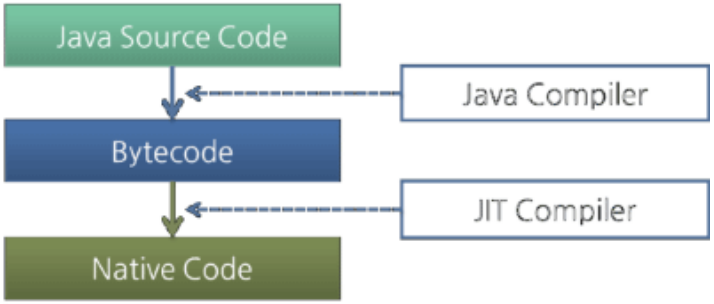
- Execution engine:

```
while(true) {
        bytecode b = bytecodeStream[pc++];
        switch(b) {
                case iconst_1: push(1); break;
                case iload_0: push(local(0)); break;
                case iadd: push(pop() + pop()); break;
        }
}
```

}NOT that simple …

The HotSpot JVM

- Different execution techniques:
  - interpreting
  - just-in-time (JIT) compilation
  - adaptive optimization (determines "hot spots" by monitoring execution)



The HotSpot JVM

- JIT compilation:
  - triggered asynchronously by counter overflow for a method/loop (interpreted counts method entries and loopback branches)
  - produces generated code and relocation info (transferred on next method entry)
  - in case JIT-compiled code calls not-yet-JIT-compiled code control is transferred to the interpreter

The HotSpot JVM

- JIT compilation:
  - compiled code may be forced back into interpreted bytecode (deoptimization)
  - is complemented by On-Stack Replacement (turn dynamically interpreted to JIT compiled code and vise-versa - dynamic optimization/deoptimization)
  - is more optimized for server VM (but hits start-up time compared to client VM)

The HotSpot JVM

- JIT compilation flow (performed during normal bytecode execution):

  1) bytecode is turned into a graph

  2) the graph is turned into a linear sequence of operations that manipulate an infinite loop of virtual registers (each node places its result in a virtual register)

The HotSpot JVM

- JIT compilation flow (performed during normal bytecode execution):

  3) physical registers are allocated for virtual registers (the program stack might be used in case virtual registers exceed physical registers)

  4) code for each operation is generated using its allocated registers

The HotSpot JVM

- Typical execution flow (when using the **java/javaw** launcher):

1. Parse the command line options
2. Establish the heap sizes and the compiler type (client or server)
3. Establish the environment variables such as CLASSPATH
4. If the java Main-Class is not specified on the command line fetch the Main-Class name from the JAR's manifest
5. Create the VM using **JNI_CreateJavaVM** in a newly created thread (non primordial thread)

The HotSpot JVM

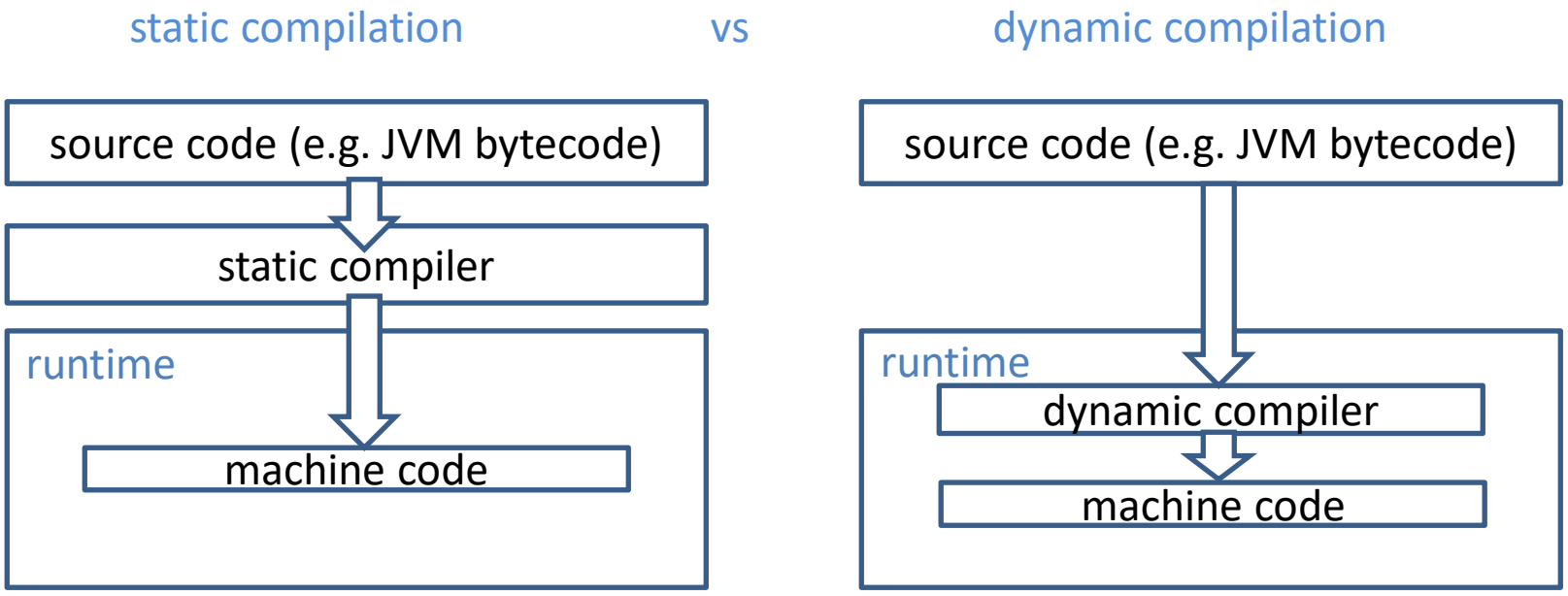
- Typical execution flow (when using the **java/javaw** launcher):

6. Once the VM is created and initialized, load the Main-Class
7. Invoke the **main** method in the VM using **CallStaticVoidMethod**
8. Once the **main** method completes check and clear any pending exceptions that may have occurred and also pass back the exit status
9. Detach the main thread using DetachCurrentThread, by doing so we decrement the thread count so the **DestroyJavaVM** can be called safely

GraalVM

GraalVM

- Some background …



GraalVM

- The JVM performs static compilation of Java sources to bytecode
- The JVM may perform dynamic compilation of bytecode to machine code for various optimizations using a JIT (Just-in-Time) compiler

GraalVM

- Graal is a new JIT (Just-in-Time) compiler for the JVM
- Brings the performance of Java to scripting languages (via the Truffle API)
- written in Java

GraalVM

- In essence the Graal JIT compiler generates machine code from an optimized AST rather than bytecode
- However the Graal VM has both AST and bytecode interpreters

GraalVM

- The Graal VM supports the following types of compilers:

```
--vm server-nograal // server compiler
--vm server // server compiler using Graal
--vm graal // Graal compiler using Graal
--vm client-nograal // client compiler
--vm client // client compiler running Graal
```

GraalVM

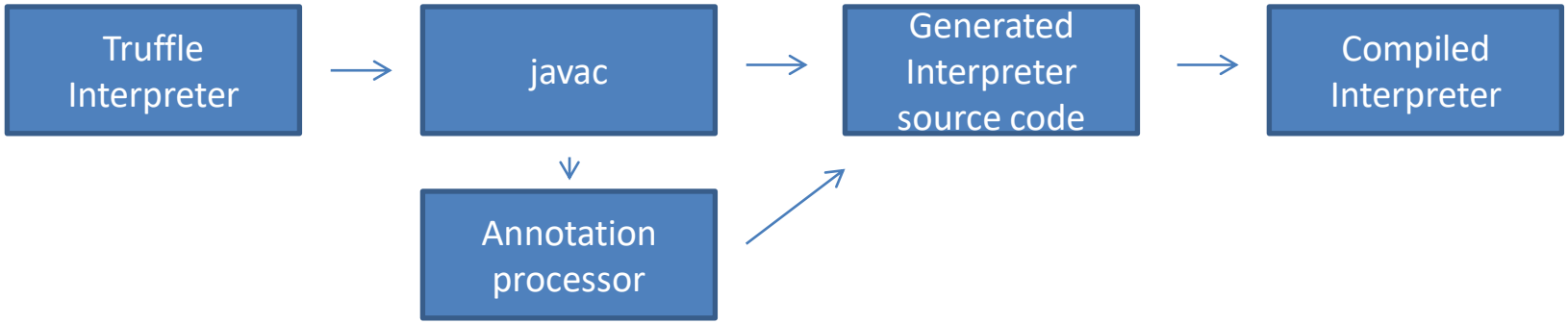
- The Truffle API:
  - is a Java API
  - provides AST (Abstract Syntax Tree) representation of source code
  - provides a mechanism to convert the generated AST into a Graal IR (intermediate representation)

GraalVM

- The Truffle API is declarative (uses Java annotations)
- The AST graph generated by Truffle is a mixture of control flow and data flow graph
- Essential feature of the Truffle API is the ability to specify node specializations used in node rewriting

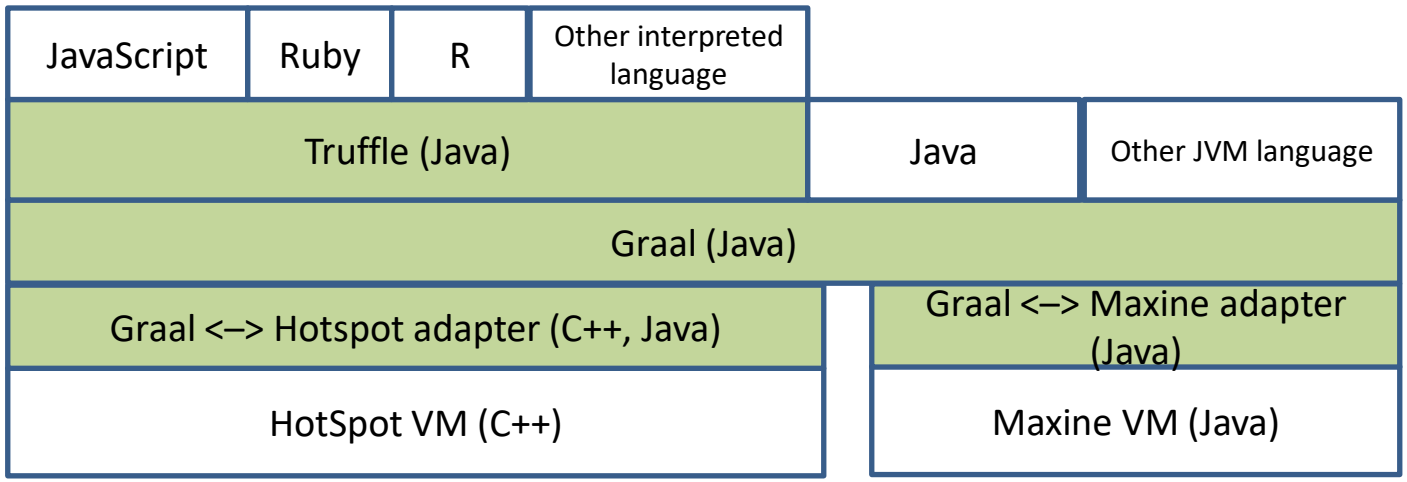
GraalVM

- The Truffle API is used in conjunction with custom annotation processor that generates code based on the Truffle annotations used in the interpreter classes



GraalVM

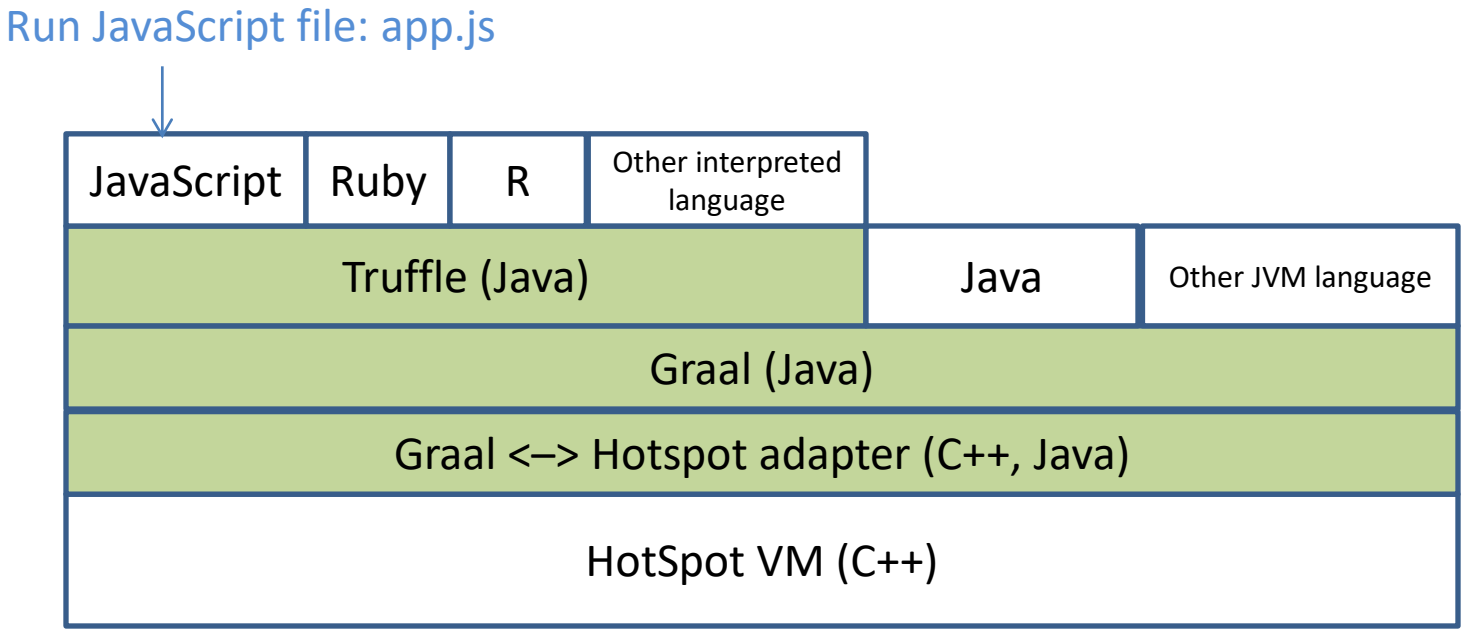
- General architecture:



GraalVM

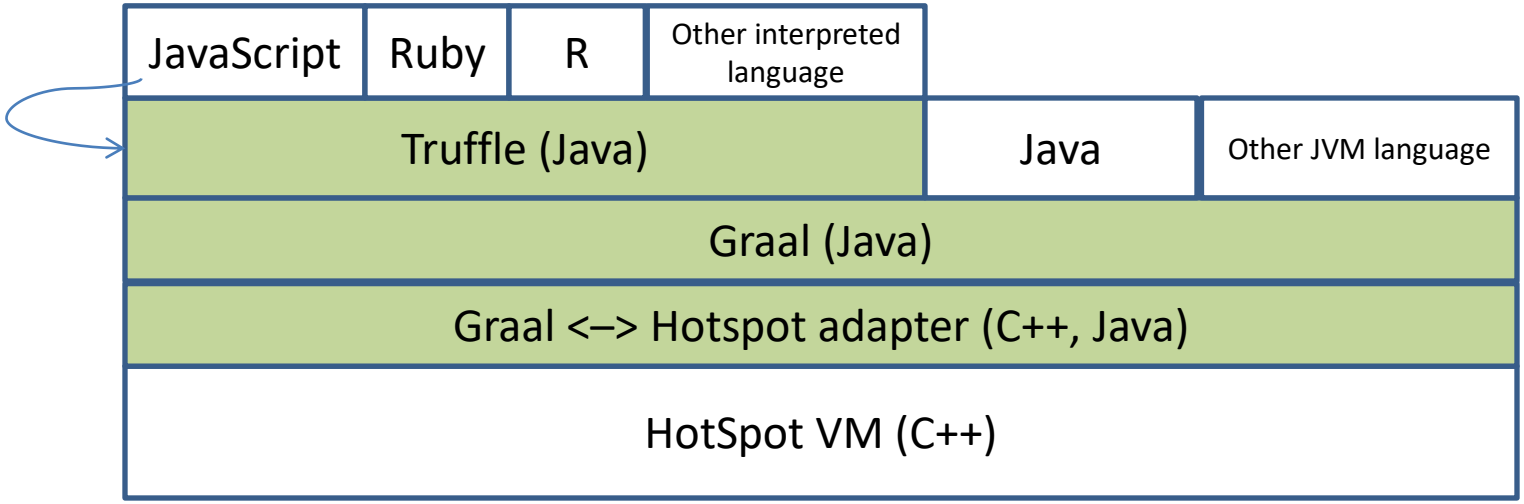
- Let’s see, for example, how a JavaScript interpreter works in Graal …

GraalVM


Run JavaScript file: app.js
JavaScript
Ruby
R
Other interpreted language
Truffle (Java)
Java
Other JVM language
Graal (Java)
Graal <–> Hotspot adapter (C++, Java)
HotSpot VM (C++)

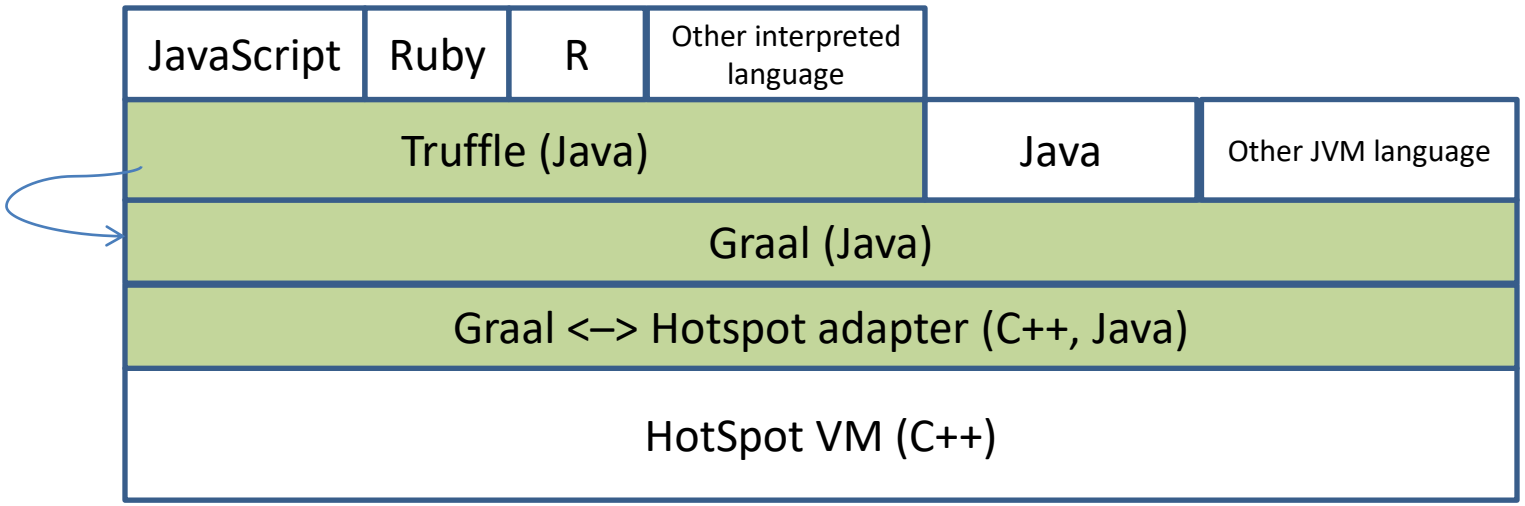
GraalVM

JavaScript Interpreter parses app.js and converts it to Truffle AST



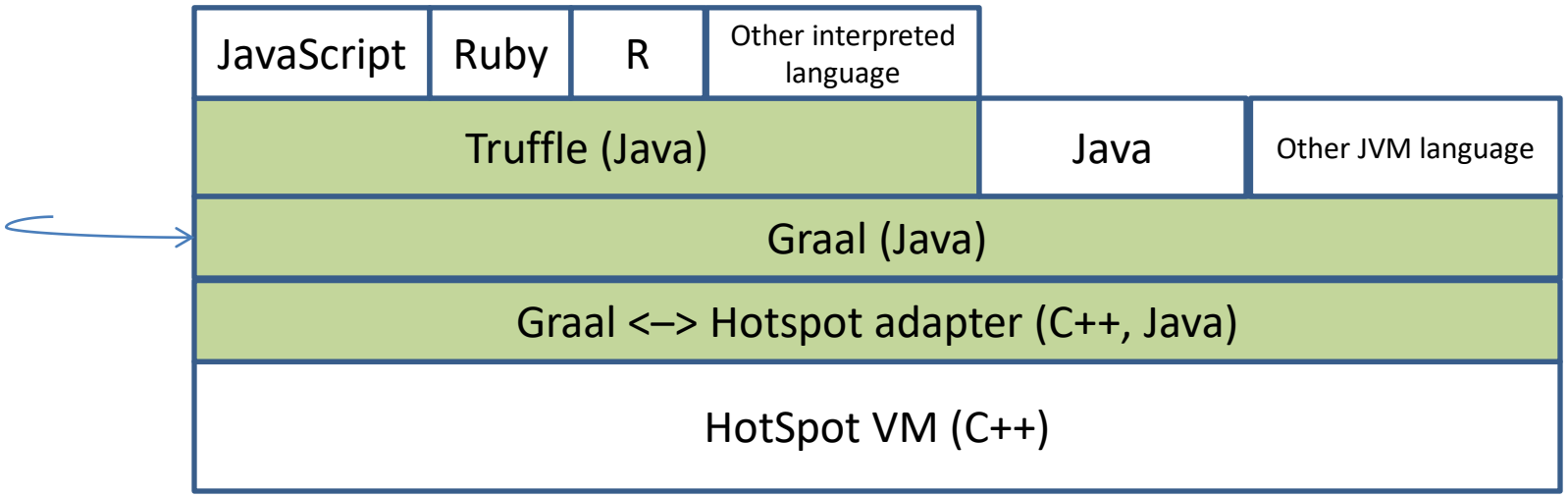
GraalVM

The app.js Truffle AST is converted to Graal IR (AST)



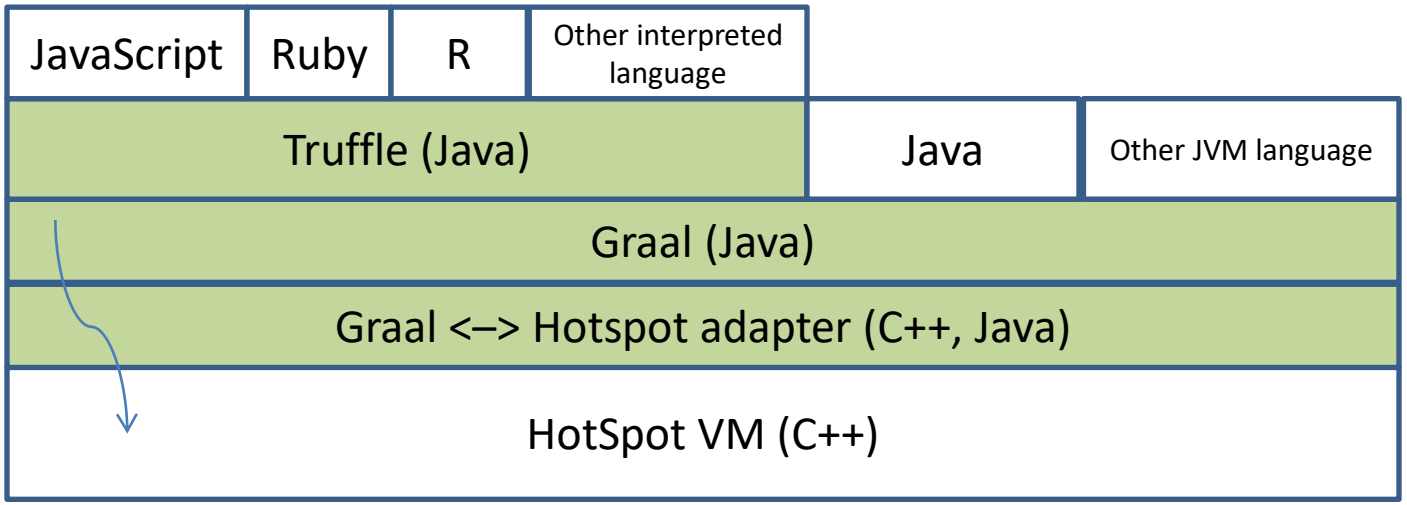
GraalVM

The Graal VM has bytecode and AST interpreters along with a JIT compiler (optimizations and AST lowering is performed to generate machine code and perform partial evaluation)



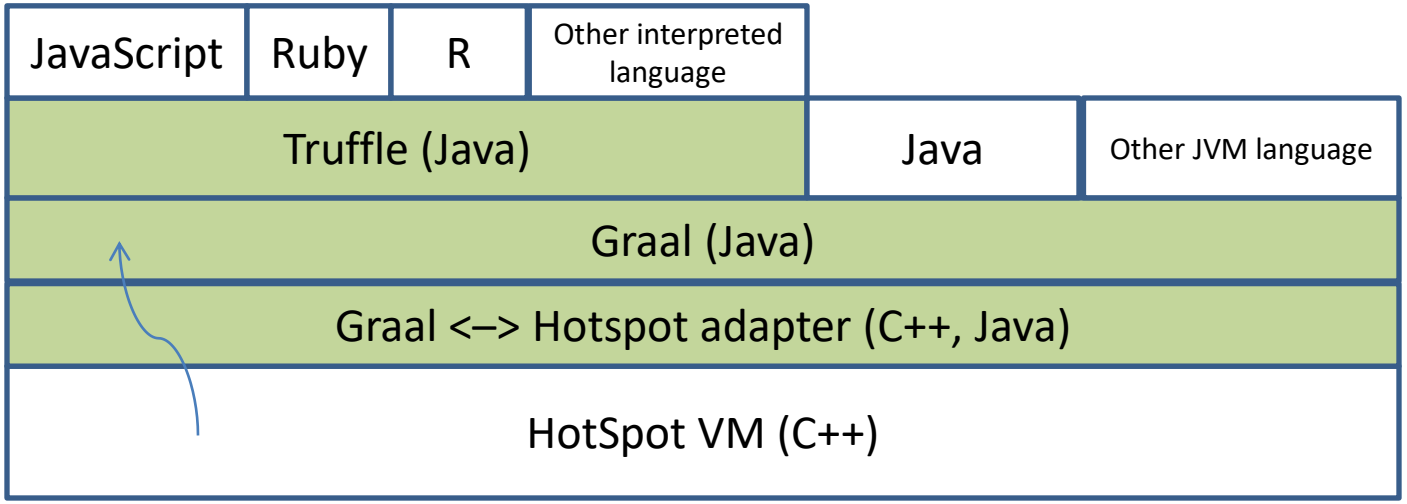
GraalVM

When parts of app.js are compiled to machine code by the Graal compiler the compiled code is transferred to Hotspot for execution by means of Hotspot APIs and with the help of a Graal – Hotspot adapter interface



GraalVM

Compiled code can also be deoptimized and control is transferred back to the interpreter (e.g. when an exception occurs, an assumption fails or a guard is reached)



GraalVM

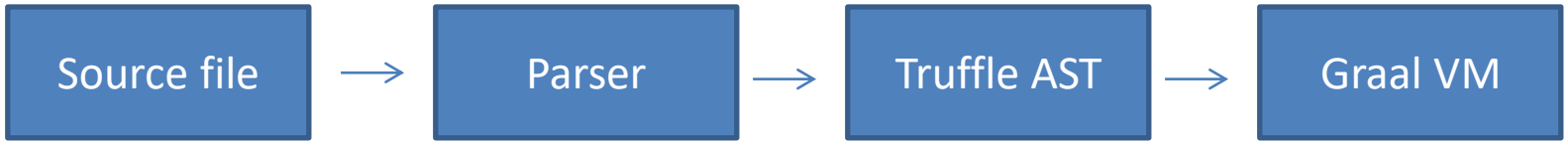
- Optimizations done on the Graal IR include:
  - method inlining
  - partial escape analysis
  - inline caching
  - constant folding
  - arithmetic optimizations and others …

GraalVM

- Currently supported languages include:
  - JavaScript (Graal.JS)
  - R (FastR)
  - Ruby (RubyTruffle)
  - Experimental interpreters for C, Python, Smalltalk, LLVM IR and others …

GraalVM

- The SimpleLanguage project provides a showcase on how to use the Truffle APIs



GraalVM

- Graal.JS – shows improvement of 1.5x in some cases with a peak of 2.6x compared to V8 (running Google Octane's benchmark)

- RubyTruffle – shows improvements between 1.5x and 4.5x in some cases with a peak of 14x compared to JRuby

- FastR - shows improvements between 2x and 39x in some cases with a peak of 94x compared to GnuR

Garbage collectors

Garbage collection

- Garbage collection is the process of scanning heap memory for unused objects and cleaning them thus reclaiming memory
- Garbage collection in the JVM is implemented by means of special modules called garbage collectors
- Garbage collectors are optimized to work over the different heap areas (generations)

Garbage collection

- Whenever the young generations (minor collections) fills up or the old generation (major collections) needs to be cleaned a “Stop The World” event appears

- In a “Stop The World” event application threads are paused during garbage collection

- In that regard garbage collection might be disruptive for performance in some high-frequency applications, i.e. in fintech

Garbage collection

- Different types of Java collectors exist at present and it is an area of active research and development:
  - Serial GC: garbage collection is done sequentially

```
-XX:-UseSerialGC
```

  - Parallel GC: uses multiple threads for garbage collection (default in JDK 7 and 8)

```
-XX:-UseParallelGC
```

```
-XX:-UseParallelOldGC
```

Garbage collection

- Different types of Java collectors exist at present and it is an area of active research and development:
  - Concurrent Mark Sweep (CMS): works concurrently along the application threads trying to minimize pauses

    ```
    -XX:-UseConcMarkSweepGC
    ```

  - G1: available as of JDK 7, parallel, concurrent and incrementally compacting low-pause collector

    (default in JDK 9 and later)

    ```
    -XX:+UseG1GC
    ```

Garbage collection

- Other types of garbage collectors also exist:
  - Epsilon GC: this is a no-op/null garbage collector introduced in JDK 11 that allocates memory but does not do garbage collection

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseEpsilonGC
```

  - ZGC: also introduced in JDK 11, still experimental, tries to reduce pause times in large scale Java applications

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseZGC
```

Garbage collection

- Other types of garbage collectors also exist:
  - Shenandoah GC: available in JDK 13 and later (also in some earlier JDK versions), aims to reduce pause times by working concurrently with the application

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseShenandoahGC
```

Questions ?