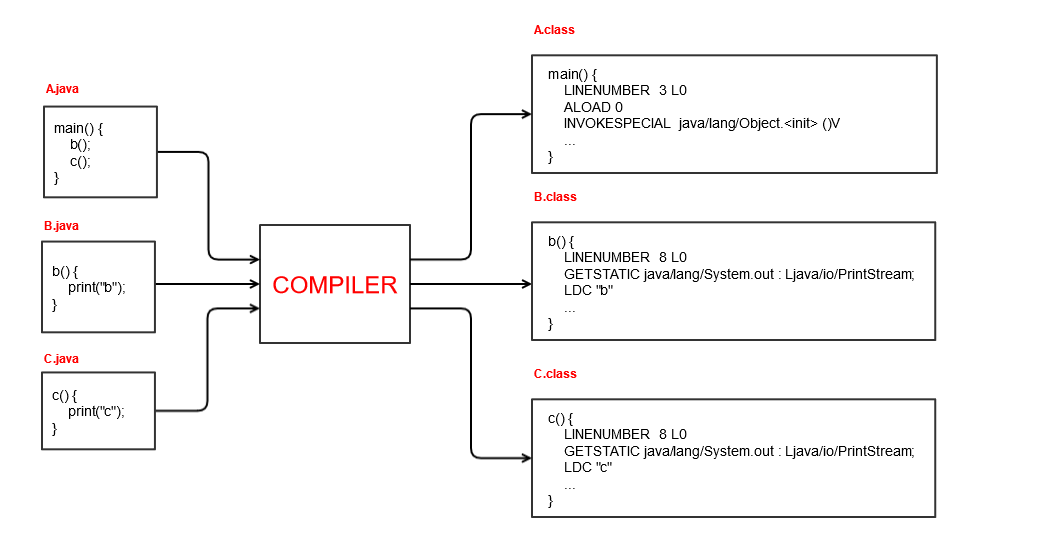
# Java compiler and interpreters with JIT and Graal

## Compiled and interpreted code

<https://www.baeldung.com/java-compiled-interpreted>

$ javac HelloWorld.java

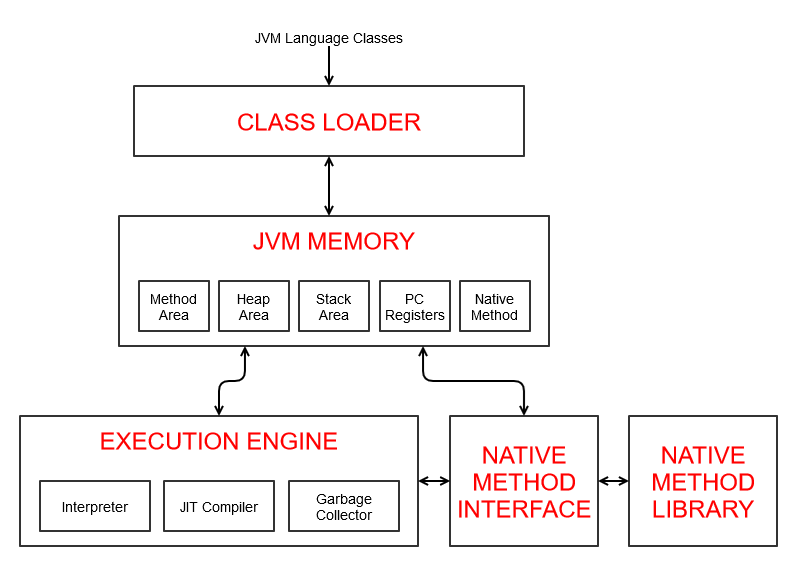
Source code files have .java suffixes, while the class files containing bytecode get generated with .class suffixes.



The compiled class files (bytecode) can be [executed](https://www.baeldung.com/java-single-file-source-code) by the [Java Virtual Machine (JVM)](https://www.baeldung.com/jvm-vs-jre-vs-jdk):

$ java HelloWorld  
Hello Java!

## Composition of the JVM



### ClassLoader

[https://www.baeldung.com/java-classloaders /](https://www.baeldung.com/java-classloaders%20/) Word file in the same folder

The JVM makes use of the [ClassLoader](https://www.baeldung.com/java-classloaders) subsystems to bring the compiled class files into [JVM memory.](https://www.baeldung.com/java-stack-heap)

Besides loading, the ClassLoader also performs linking and initialization. That includes:

* Verifying the bytecode for any security breaches
* Allocating memory for static variables
* Replacing symbolic memory references with the original references
* Assigning original values to static variables
* Executing all static code blocks

### JVM memory

<https://www.baeldung.com/java-stack-heap> /Word file

### Execution Engine

The execution engine subsystem is in charge of **reading the bytecode, converting it into machine native code, and executing it.**

Three major components are in charge of execution, including both an interpreter and a compiler:

Since the JVM is platform-neutral, it uses an **interpreter** to execute bytecode

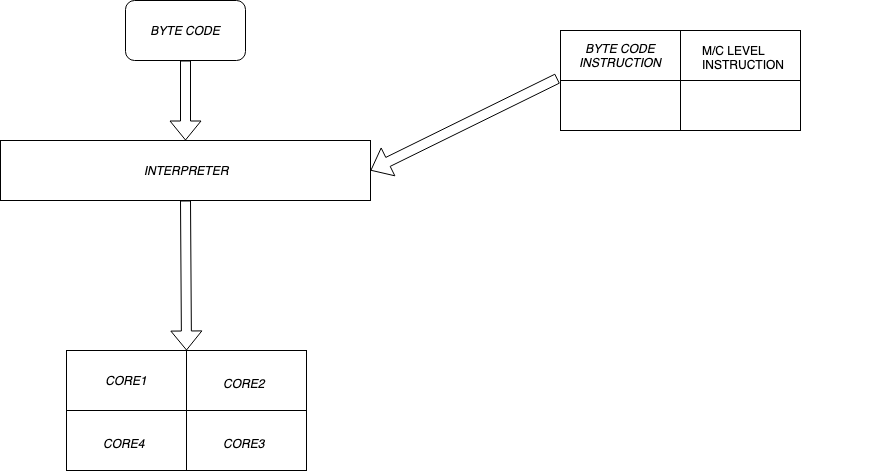
The [**JIT compiler**](https://www.baeldung.com/ahead-of-time-compilation) improves performance by compiling bytecode to native code for repeated method calls.

The [**Garbage collector**](https://www.baeldung.com/jvm-garbage-collectors) collects and removes all unreferenced objects

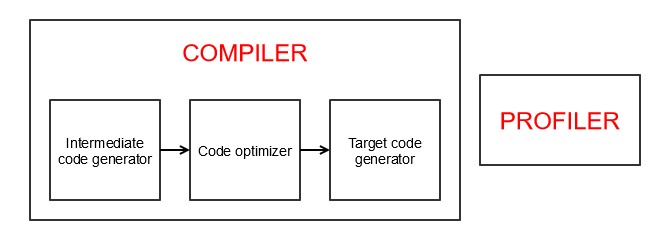
The execution engine makes use of the [Native method interface (JNI)](https://www.baeldung.com/jni) to call native libraries and applications.

### The interpreter

The **interpreter** keeps a table that converts bytecode instruction in machine instructions (jvm/machine dependent). It does it for each instruction without any optimization. This is done by the C1 and C2 compilers



A **profiler** is a special component of the JIT compiler responsible for finding hotspots. The JVM decides which code to JIT compile based on the profiling information collected during runtime.

[](https://www.baeldung.com/wp-content/uploads/2021/01/jit_compiler1.png)

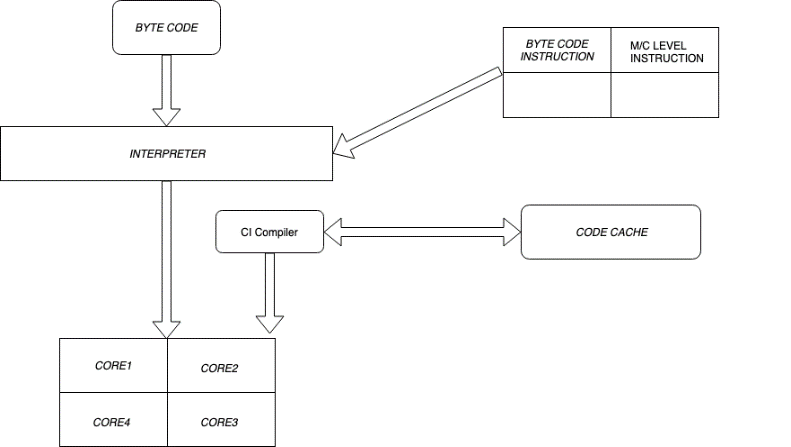
One effect of this is that a Java program can become faster at performing its job after a few cycles of execution. Once the JVM has learned the hotspots, it is able to create the native code allowing things to run faster.

### The HotSpot virtual machine

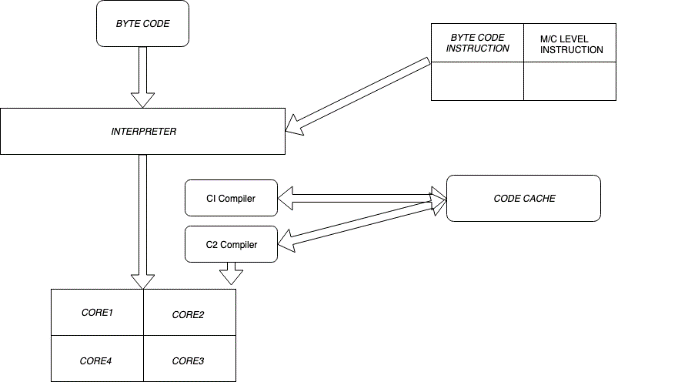
The JDK implementation by Oracle is based on the open-source OpenJDK project. This includes the **HotSpot virtual machine**, available since Java version 1.3. It **contains two conventional JIT-compilers: the client compiler, also called C1 and the server compiler, called opto or C2**.

C1 is designed to run faster and produce less optimized code, while C2, on the other hand, takes a little more time to run but produces a better-optimized code. The client compiler is a better fit for desktop applications since we don't want to have long pauses for the JIT-compilation. The server compiler is better for long-running server applications that can spend more time on the compilation.

### The client JIT compiler C1

The c1 compiler keeps a count the number of calls to a certain method bytecode is used. It will be cached if the count goes over a certain threshold (1500 by default). Code cache is a memory area separate from the JVM heap that contains all the JVM bytecode for a method compiled down to native code, each called a **nmethod1**. This is where the JIT compiled methods are kept.

### The server JIT compiler C2

The C2 compiler runs in parallel with the C1 and uses uses this PD to store to make certain assumptions such as what code paths are cold/warm/hot, and what types are used at any call sites. This will determine whether a bytecode will be stored in the cache. It can then generate code better suited for the specific context that it is currently executing in.

### Just in Time Compiler modes: Tiered compilation

Introduced in Java 7, **Tiered Compilation** (TC) goal is to have fast startup time and fast steady-state throughput. The implementation consists of a pipeline of multiple tiers of code generation. The three main components of this pipeline are the **interpreter**, the **C1 compiler**, and the **C2 compiler**. It replaced the -client and -server command-line parameters available in previous versions of Java.

**The JIT compiler compiles the entire method's bytecode to machine native code**, so it can be reused directly**.**As with a standard compiler, there's the generation to intermediate code, optimization, and then the production of machine native code.

As the method goes through the different tiers, each tier gathers information about the method execution. This information is called **Profiling Data** (PD). The C2 compiler uses this PD to make certain assumptions such as what code paths are cold/warm/hot, and what types are used at any call sites. It can then generate code better suited for the specific context that it is currently executing in.

The five tiers of code generation are:

**none (0):** Interpreter gathering full PD

**simple (1):** C1 compiler with no profiling

**limited profile (2):** C1 compiler with light profiling gathering some PD

**full profile (3):** C1 compiler with full profiling gathering full PD

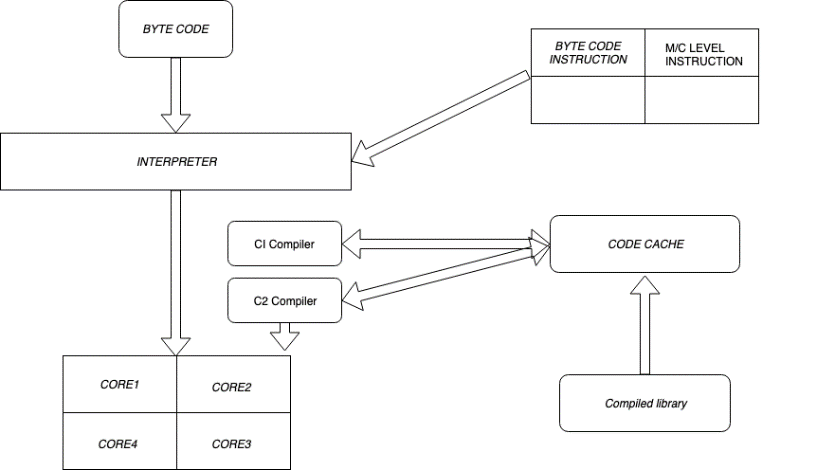
**full optimization (4):** C2 compiler with no profiling

Two compilers, C1, and C2 run in parallel and keep on optimizing the code

C1 is preferred for the client application and C2 is preferred for long running server applications.

Tiered compilation combines the best features of both compilers. Client-side compilation yields quick startup time and speedy optimization, while server-side compilation delivers more advanced optimizations later in the execution cycle.

When the Code Cache is constrained (its usage approaches or reaches the ReservedCodeCacheSize), to compile more methods, the JIT must first throw out some already compiled methods. Discarding compiled methods are known as Code Cache flushing.

In JAVA 7 we have the option to select to both the compiler. In JAVA 8 both are available by default.

### Ahead of time compilation

After Java9 we have the option to convert some of the classes or libraries to compiled code before the start of the application.

#### Usage of the AOT

For the AOT compiler to successfully generate code, the same environment than for the JIT compiler need to be available. That means that all dependencies (jars, jmods) must be present and accessible to the AOT compiler.

*The example below is assuming you are using Java 11 or later.*

class HelloWorld {

public static void main(String args[]) {

System.out.println("Hello, World");

}

}

To compile it to Java bytecode, run the usual:

$> javac HelloWorld.java

If you want to run without AOT, you simply run:

$> java HelloWorld

Hello, World

If you want to run with AOT, you first need to run the AOT compiler:

$> jaotc --compile-for-tiered --output libHelloWorld.so --verbose HelloWorld

Compiling libHelloWorld.so...

1 classes found (25 ms)

Scanning HelloWorld

added <init>()V

added main([Ljava/lang/String;)V

2 methods total, 2 methods to compile (4 ms)

Freeing memory [used: 4.0 MB , comm: 12.0 MB, freeRatio ~= 66.7%] (44 ms)

Compiling with 12 threads

…

Total time: 911 ms

Then to reference the code generated by the AOT compiler, run:

$> java -XX:AOTLibrary=./libHelloWorld.so HelloWorld

Hello, World

To verify if the AOT compiled code is loaded and executed, run the above command with -XX:+PrintAOT and you should observe the following output:

$> java -XX:AOTLibrary=./libHelloWorld.so -XX:+PrintAOT HelloWorld

17    1     loaded    ./libHelloWorld.so  aot library

58    1     aot[ 1]   HelloWorld.<init>()V

58    2     aot[ 1]   HelloWorld.main([Ljava/lang/String;)V

Hello, World

You can observe the output of -XX:PrintAOT in the first three lines. Line 1 signals that ./libHelloWorld.so was correctly loaded. Lines 2 and 3 signal that the constructor HelloWorld.<init>() and the main method HelloWorld.main() were loaded and used for this execution of the application.

The AOT compiled library contains a class ***fingerprint***, which must match the fingerprint of the .class file.If we change the method to print Ciao, Mondo instead. We can still run the program without the OAT support for this change:

$> java -XX:AOTLibrary=./libHelloWorld.so -XX:+PrintAOT HelloWorld

17    1     loaded    ./libHelloWorld.so  aot library

Hello, World

**We can see that the methods in the library won't be called, as the bytecode of the class has changed.** The idea behind this is that the program will always produce the same result, no matter if an AOT compiled library is loaded or not.

## Performance Comparison

Let's take a look at how the JIT compilation improves Java's runtime performance.

### Fibonacci Performance Test

We'll use a simple recursive method to calculate the n-th Fibonacci number:

**private** **static** **int** **fibonacci**(**int** index) {

**if** (index <= 1) {

**return** index;

}

**return** fibonacci(index-1) + fibonacci(index-2);

}

In order to measure performance benefits for repeated method calls, we'll run the Fibonacci method 100 times:

**for** (**int** i = 0; i < 100; i++) {

**long** startTime = System.nanoTime();

**int** result = fibonacci(12);

**long** totalTime = System.nanoTime() - startTime;

System.out.println(totalTime);

}

First, we'll compile and execute the Java code normally:

$ java Fibonacci.java

Then, we'll execute the same code with the **JIT compiler disabled**:

$ java -Djava.compiler=NONE Fibonacci.java

Finally, we'll implement and run the same algorithm in C++ and JavaScript for comparison.

### Performance Test Results

Let's take a look at the measured average performances in nanoseconds after running the Fibonacci recursive test:

* Java using JIT compiler – 2726 ns – fastest
* Java without JIT compiler  –  17965 ns – 559% slower
* C++ without O2 optimization –  9435 ns – 246% slower
* C++ with O2 optimization –  3639 ns – 33% slower
* JavaScript –  22998 ns – 743% slower

In this example, **Java's performance is more than 500% better using the JIT compiler**. However, it does take a few runs for the JIT compiler to kick-in.

Interestingly, Java performed 33% better than C++ code, even when C++ is compiled with the O2 optimization flag enabled. As expected, **C++ performed much better in the first few runs**, when Java was still interpreted.

Java also outperformed the equivalent JavaScript code run with Node, which also uses a JIT compiler. Results show more than 700% better performance. The main reason is that **Java's JIT compiler kicks-in much faster.**

## Things to Consider

Technically, it's possible to compile any static programming language code to machine code directly. It's also possible to interpret any programming code step-by-step.

Similar to many other modern programming languages, Java uses a combination of a compiler and interpreter. The goal is to make use of the best of both worlds, **enabling high performance and platform-neutral execution**.

In this article, we focused on explaining how things work in HotSpot. **HotSpot** is the default open-source JVM implementation by Oracle. [Graal VM](https://www.baeldung.com/graal-java-jit-compiler) is also based on HotSpot, so the same principles apply.

Most popular JVM implementations nowadays use a **combination of an interpreter and a JIT compiler.** However, it's possible that some of them use a different approach.

# The new Java JIT Compiler - GraalVM

<https://www.baeldung.com/graal-java-jit-compiler>

Project GraalVM is a research project created by Oracle. We can look at Graal as several connected projects: **a new JIT compiler, written in JAVA, that builds on HotSpot** and **a new polyglot virtual machine**. It offers a comprehensive ecosystem supporting a large set of languages (Java and other JVM-based languages; JavaScript, Ruby, Python, R, C/C++, and other LLVM-based languages).

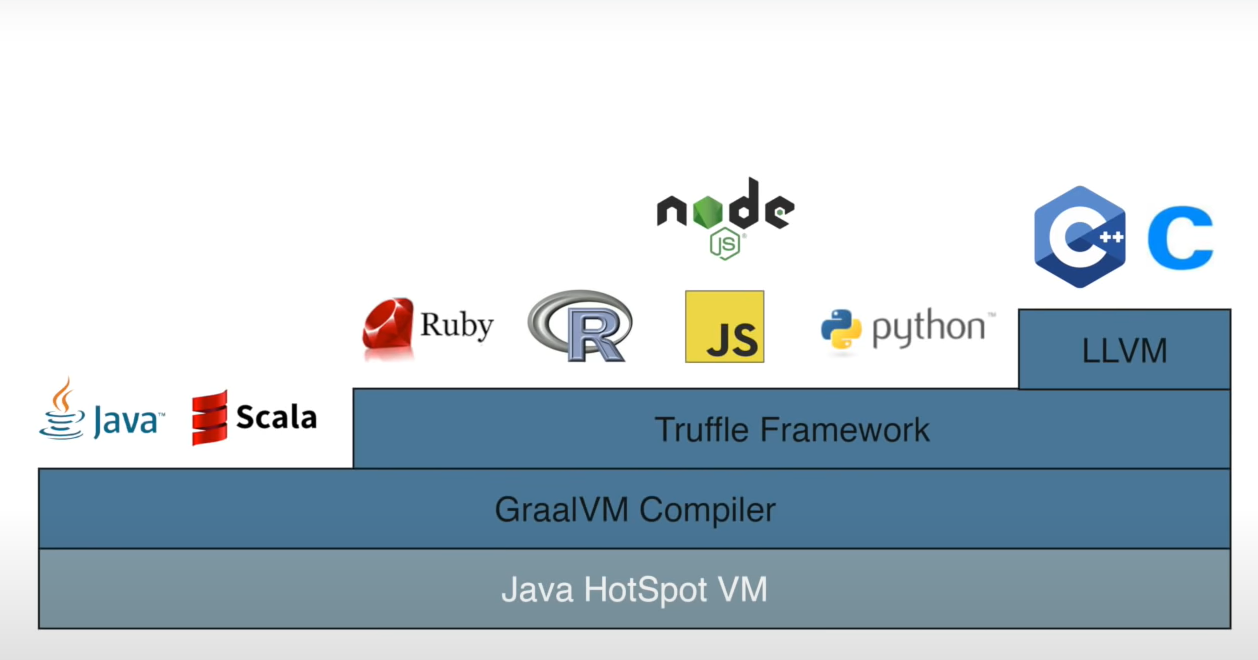
**Graal is a high-performance JIT compiler.**It accepts the JVM bytecode and produces the machine code. There are several key advantages of writing a compiler in Java. First of all, **safety**, meaning no crashes but exceptions instead and no real memory leaks. Furthermore, we'll have a **good IDE support** and we'll be able to use debuggers or profilers or other convenient tools. Also, **the compiler can be independent of the HotSpot** and it would be able to produce a faster JIT-compiled version of itself.

The Graal compiler was created with those advantages in mind. It uses the new **JVM Compiler Interface – JVMCI** to communicate with the VM. To enable the use of the new JIT compiler, we need to set the following options when running Java from the command line:

-XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler

What this means is that we can run a simple program in three different ways: with the regular tiered compilers, with the JVMCI version of Graal on Java 10 or with the GraalVM itself.

## Introduction to the GraalVM architecture



GraalVM adds an advanced just-in-time (JIT) optimizing compiler, which is written in Java, to the HotSpot Java Virtual Machine. The **GraalVM Compiler** substitutes the standard JIT compilers (C2) and implements the JVMCI (Java Virtual Machine Command Interface, introduced in Java 9) to talk to the underlying implementation (**JAVA HotSpot VM**). You can still activate the C2 compiler through configuration of the VM. This can run all JVM based languages (JAVA, Scala).

### Truffle Language Implementation Framework

If we want to support other languages (JS, Ruby, Python…), we have to translate all the language source into bytecode in order to be run in the JVM. This is a maintenance nightmare. Then the **Language Implementation Framework Truffle** has been created to create interpreters for the different languages, which is way easier than building a compiler for that language. In other words, by using the Truffle Language Implementation Framework you can easily implement any language based on an <https://en.wikipedia.org/wiki/Abstract_syntax_tree>. And run all of them in the same runtime environment sharing the same memory.

In addition to running Java and JVM-based languages, GraalVM’s Truffle language implementation framework makes it possible to run JavaScript, Ruby, Python, and a number of other popular languages on the JVM. With GraalVM Truffle, Java and other supported languages can directly interoperate with each other and pass data back and forth in the same memory space. In addition to that a LLVM compiler can be used on top of Truffle for C or C++ code.

At runtime all code interpreted is a tree of nodes, no matter what language was use originally, so you can inline native extension code with dynamic code and interoperate with them.

The Truffle language implementation framework (henceforth “Truffle”) is an open source library for building tools and programming languages implementations as interpreters for self-modifying Abstract Syntax Trees. Together with the open source GraalVM compiler, Truffle represents a significant step forward in programming language implementation technology in the current era of dynamic languages.

### JVM Compiler Interface

The JVMCI is part of the OpenJDK since JDK 9, so we can use any standard OpenJDK or Oracle JDK to run Graal.

**What JVMCI actually allows us to do is to exclude the standard tiered compilation and plug in our brand new compiler (i.e. Graal) without the need of changing anything in the JVM.**

The interface is quite simple. When Graal is compiling a method, it'll pass the bytecode of that method as the input to the JVMCI'. As an output, we'll get the compiled machine code. Both the input and the output are just byte arrays:

interface JVMCICompiler {

byte[] compileMethod(byte[] bytecode);

}

In real-life scenarios, we'll usually need some more information like the number of local variables, the stack size, and the information collected from profiling in the interpreter so that we know how the code is running in practice.

Essentially, when calling the **compileMethod()** of the *[JVMCICompiler](https://github.com/md-5/OpenJDK/blob/master/src/jdk.internal.vm.ci/share/classes/jdk.vm.ci.runtime/src/jdk/vm/ci/runtime/JVMCICompiler.java)* interface, we'll need to pass a **CompilationRequest** object. It'll then return the Java method we want to compile, and in that method, we'll find all the information we need.

### Graal in Action

Graal itself is executed by the VM, so it'll first be interpreted and JIT-compiled when it becomes hot. Let's check out an example, which can be also found on the [GraalVM's official site](https://www.graalvm.org/docs/examples/java-performance-examples/):

**public** **class** **CountUppercase** {

**static** **final** **int** ITERATIONS = Math.max(Integer.getInteger("iterations", 1), 1);

**public** **static** **void** **main**(String[] args) {

String sentence = String.join(" ", args);

**for** (**int** iter = 0; iter < ITERATIONS; iter++) {

**if** (ITERATIONS != 1) {

System.out.println("-- iteration " + (iter + 1) + " --");

}

**long** total = 0, start = System.currentTimeMillis(), last = start;

**for** (**int** i = 1; i < 10\_000\_000; i++) {

total += sentence

.chars()

.filter(Character::isUpperCase)

.count();

**if** (i % 1\_000\_000 == 0) {

**long** now = System.currentTimeMillis();

System.out.printf("%d (%d ms)%n", i / 1\_000\_000, now - last);

last = now;

}

}

System.out.printf("total: %d (%d ms)%n", total, System.currentTimeMillis() - start);

}

}

}

Now, we'll compile it and run it:

javac CountUppercase.java

java -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler

This will result in the output similar to the following:

1 (1581 ms)

2 (480 ms)

3 (364 ms)

4 (231 ms)

5 (196 ms)

6 (121 ms)

7 (116 ms)

8 (116 ms)

9 (116 ms)

total: 59999994 (3436 ms)

We can see that **it takes more time in the beginning**. That warm-up time depends on various factors, such as the amount of multi-threaded code in the application or the number of threads the VM uses. If there are fewer cores, the warm-up time could be longer.

If we want to see the statistics of Graal compilations we need to add the following flag when executing our program:

-Dgraal.PrintCompilation=true

This will show the data related to the compiled method, the time taken, the bytecodes processed (which includes inlined methods as well), the size of the machine code produced, and the amount of memory allocated during compilation. The output of the execution takes quite a lot of space, so we won't show it here.

### Comparing with the Top Tier Compiler

Let's now compare the above results with the execution of the same program compiled with the top tier compiler instead. To do that, we need to tell the VM to not use the JVMCI compiler:

java -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:-UseJVMCICompiler

1 (510 ms)

2 (375 ms)

3 (365 ms)

4 (368 ms)

5 (348 ms)

6 (370 ms)

7 (353 ms)

8 (348 ms)

9 (369 ms)

total: 59999994 (4004 ms)

We can see that there is a smaller difference between the individual times. It also results in a briefer initial time.

### Escape analysis

<https://www.javaadvent.com/2020/12/seeing-escape-analysis-working.html>

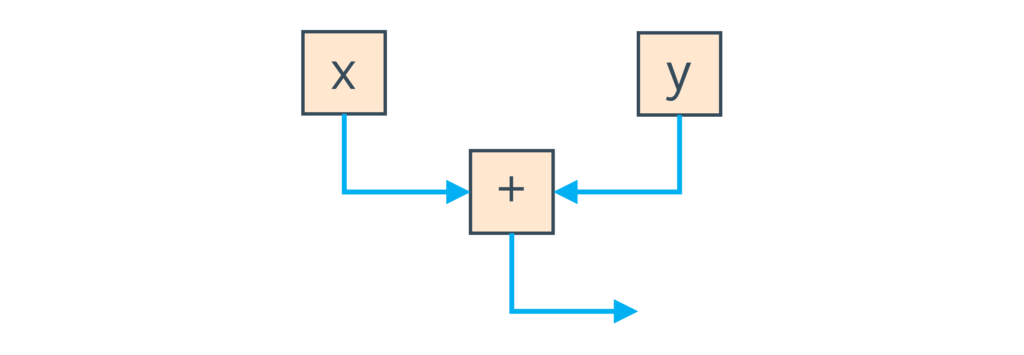
This feature of the Graal compiler is very useful while working with Immutable data structures (streams, immutable collections). Those immutable entities are very intuitive but they have a drawback: every operation on them will allocate a new instance (as with String). This might generate a huge amount of unwilled memory allocations at runtime (very slow op) useless if the intermediate allocations are not propagated outside the current *compilation unit* (a method and the inline other method calls). The escape analysis removes the intermediate allocations with the corresponding scalar values of the fields handled at runtime. **This means that the runtime program will not allocate much memory resulting in less pressure on the garbage collector**.

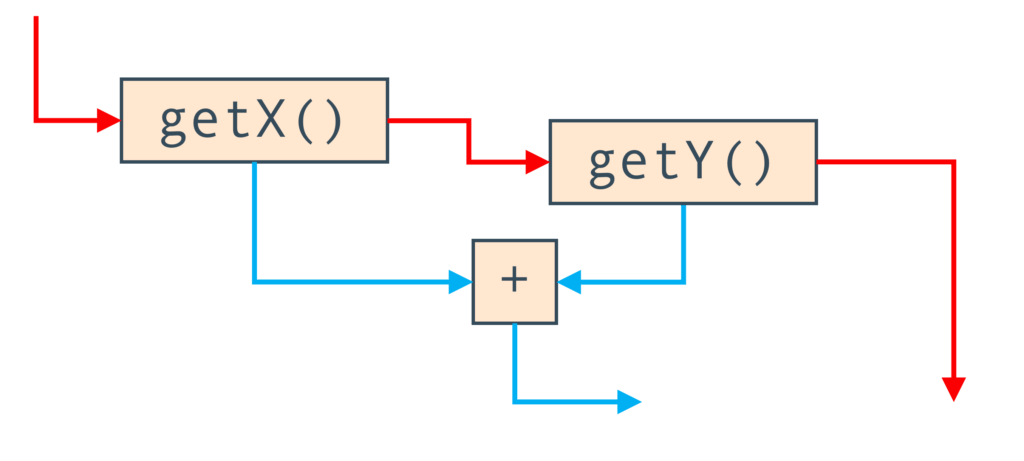
### The Data Structure Behind Graal

As we said earlier, Graal basically turns a byte array into another byte array. In this section, we'll focus on what's behind this process. The following examples are relying on [Chris Seaton's talk at JokerConf 2017](https://chrisseaton.com/truffleruby/jokerconf17/).

Basic compiler's job, in general, is to act upon our program. This means that it must symbolize it with an appropriate data structure. Graal uses a graph for such a purpose, the so-called **program-dependence-graph**.

In a simple scenario, where we want to add two local variables, i.e., x + y, we would have one node for loading each variable and another node for adding them. Beside it, we'd also have two edges representing the data flow:

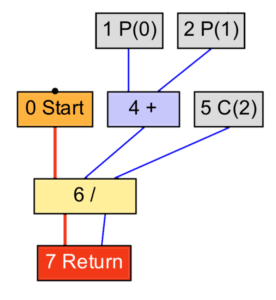
[](https://www.baeldung.com/wp-content/uploads/2018/11/data-graph-x-p-y.png)**The data flow edges are displayed in blue**. They're pointing out that when the local variables are loaded, the result goes into the addition operation.

[](https://www.baeldung.com/wp-content/uploads/2018/11/control-graph-getx-p-gety.png)Let's now introduce another type of edges, the **control flow edges in red**. To do so, we'll extend our example by calling methods to retrieve our variables instead of reading them directly. When we do that, we need to keep track of the methods calling order. We'll represent this order with the red arrows:

Here, we can see that the nodes didn't change actually, but we have the control flow edges added.

### Actual Graphs

We can examine the real Graal graphs with the [IdealGraphVisualiser](http://ssw.jku.at/General/Staff/TW/igv.html). To run it, we use the **mx igv** command. We also need to configure the JVM by setting the **-Dgraal.Dump** flag.

[](https://www.baeldung.com/wp-content/uploads/2018/11/graph-average.png)Let's check out a simple example:

**int** **average**(**int** a, **int** b) {

**return** (a + b) / 2;

}

This has a very simple data flow:

In the graph above, we can see a clear representation of our method. Parameters P(0) and P(1) flow into the add operation which enters the divide operation with the constant C(2). Finally, the result is returned.

We'll now change the previous example to be applicable to an array of numbers:

**int** **average**(**int**[] values) {

**int** sum = 0;

**for** (**int** n = 0; n < values.length; n++) {

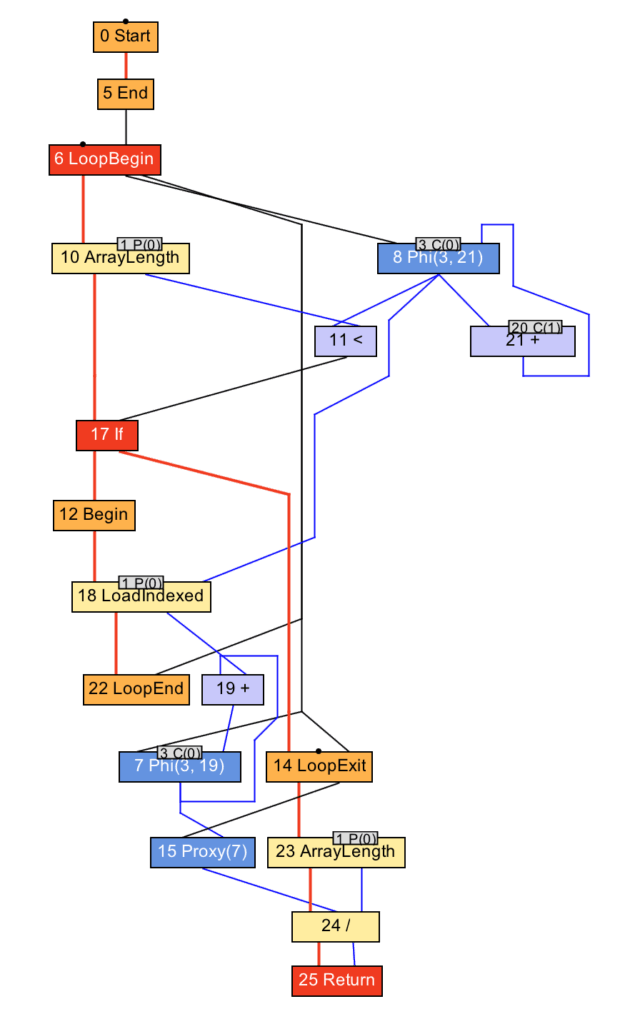
sum += values[n];

}

**return** sum / values.length;

}

We can see that adding a loop led us to the much more complex graph:

[](https://www.baeldung.com/wp-content/uploads/2018/11/average-loop-detail.png)

What we can notice here are**:**

* the begin and the end loop nodes
* the nodes representing the array reading and the array length reading
* data and control flow edges, just as before.

**This data structure is sometimes called a sea-of-nodes, or a soup-of-nodes**. We need to mention that the C2 compiler uses a similar data structure, so it's not something new, innovated exclusively for Graal.

It is noteworthy remember that Graal optimizes and compiles our program by modifying the above-mentioned data structure. We can see why it was an actually good choice to write the Graal JIT compiler in Java: **a graph is nothing more than a set of objects with references connecting them as the edges. That structure is perfectly compatible with the object-oriented language, which in this case is Java**.

### Ahead-of-Time Compiler Mode

It is also important to mention that **we can also use the Graal compiler in the Ahead-of-Time compiler mode in Java 10**. As we said already, the Graal compiler has been written from scratch. It conforms to a new clean interface, the JVMCI, which enables us to integrate it with the HotSpot. That doesn't mean that the compiler is bound to it though.

One way of using the compiler is to use a profile-driven approach to compile only the hot methods, but **we can also make use of Graal to do a total compilation of all methods in an offline mode without executing the code**. This is a so-called “Ahead-of-Time Compilation”, [JEP 295,](https://openjdk.java.net/jeps/295) but we'll not go deep into the AOT compilation technology here.

The main reason why we would use Graal in this manner is to speed up startup time until the regular Tiered Compilation approach in the HotSpot can take over.

## Runtime Components

GraalVM is unique as a runtime environment offering several components: JVM runtime mode, Native Image, Java on Truffle (the same Java applications can be run on either).

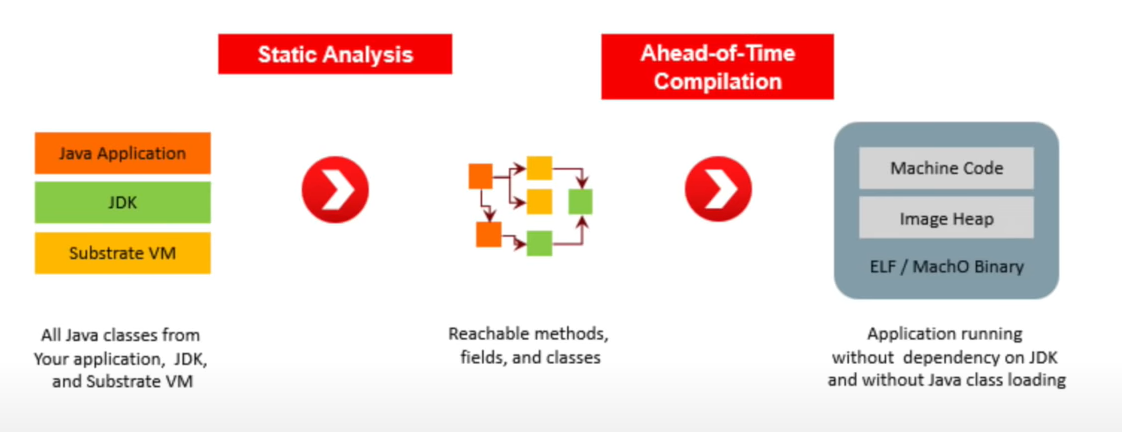
### JVM Runtime Mode

When running programs on the HotSpot JVM, GraalVM defaults to the [GraalVM compiler](https://www.graalvm.org/reference-manual/compiler/) as the top-tier JIT compiler It complements or replaces the existing compilers (C1/C2 in [HotSpot](https://en.wikipedia.org/wiki/HotSpot_(virtual_machine)" \o "HotSpot (virtual machine))). In contrast to those existing compilers, the GraalVM compiler is written in modular, maintainable and extendable fashion in Java itself. At runtime, an application is loaded and executed normally on the JVM. The JVM passes bytecodes for Java or any other JVM-native language to the compiler, which compiles that to the machine code and returns it to the JVM. Interpreters for supported languages, written on top of the [Truffle framework](https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/), are themselves Java programs that run on the JVM.

### Native Image

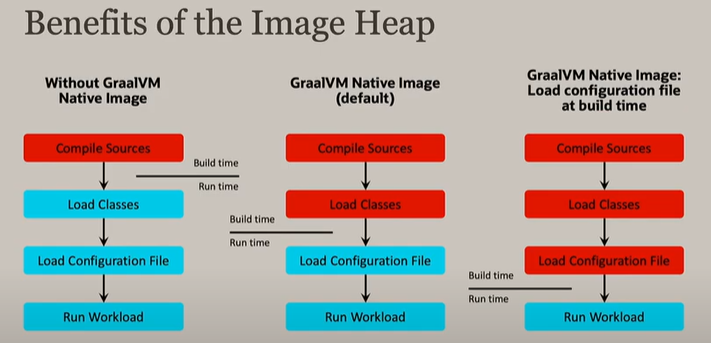
https://www.youtube.com/watch?v=WKrR4PGTgoc&t=190s

[Native Image](https://www.graalvm.org/reference-manual/enterprise-native-image/) is an innovative technology that compiles Java code into a standalone binary executable or a native shared library (see JNDI document). The Java bytecode that is processed during the native image build includes all application classes, dependencies, third party dependent libraries, and any JDK classes that are required. This runs under the **close world assumption**: meaning that all accessible (byte)code should be reachable at compilation time, in order to do more complete and better optimization of the generated image. A generated self-contained native executable is specific to each individual operating systems and machine architecture that does not require a JVM.



The static analysis step do not execute the code but goes through all the classes that are involved (also in the called libraries), making the optimizations and the binary conversion (with just a wrapper to allow the OS to execute it).

#### Image heap



#### Required configuration

This is not easy in case of usage of **reflection** in the code. A program could be totally based on user input to identify the methods to run through reflection… In such a case no static analysis can predict what the program will do at runtime.

For that you need to pass in some configuration (json configuration) also for proxy JNI configuration, for the usage of reflection we do in our application. We can also use a framework to generate those configuration files, the **native-image-agent**.

### Java on Truffle

[Java on Truffle](https://www.graalvm.org/reference-manual/java-on-truffle/) is an implementation of the Java Virtual Machine Specification, built with the [Truffle language implementation framework](https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/). It is a complete Java VM that includes all core components, implements the same API as the Java Runtime Environment library, and reuses all JARs and native libraries from GraalVM. Java on Trufle is an experimental technology in GraalVM, available as of version 21.0.0. Now Java can be executed by the same principle as other languages in the GraalVM ecosystem (JavaScript, Ruby, Python, R), directly interoperate with those languages, and pass data back and forth in the same memory space. Besides complete language interoperability, with Java on Truffle you can:

* run Java bytecodes in a separate context from the host Java VM. It can run either a Java 8 or Java 11 guest or host JVM. In other words, you can embed a Java 8 context in a Java 11 application, by using [GraalVM’s Polyglot API](https://www.graalvm.org/sdk/javadoc/org/graalvm/polyglot/package-summary.html).
* leverage the whole stack of tools provided by the Truffle framework, not previously available for Java.
* have an improved isolation of the host Java VM and the Java program running on Truffle, so you can run less trusted guest code.
* run in the context of a native image while still allowing dynamically-loaded bytecodes.

Java on Trufle is an experimental technology in GraalVM, but already passes the Java Compatibility Kit (JCK or TCK for Java SE). It is available as of version 21.0.0.

https://medium.com/graalvm/graalvm-ten-things-12d9111f307d

We'll of course focus on Java.

## Spring boot support for GraalVm native image

https://www.infoq.com/news/2021/03/spring-native-beta-available/