**GRAAL DEEP DIVE**

1. **Building and running an application using the Graal compiler**

Steps:-

1/ Checkout https://github.com/oracle/graal.git

2/ Checkout https://github.com/graalvm/mx.git

3/ set <dir>/mx to PATH

4/ cd <dir>/graal/compiler

mx --java-home <JDK-26> build

mx --java-home <JDK-26> archive // optional

cd ../<dir>/vm

mx --java-home <JDK-26> build

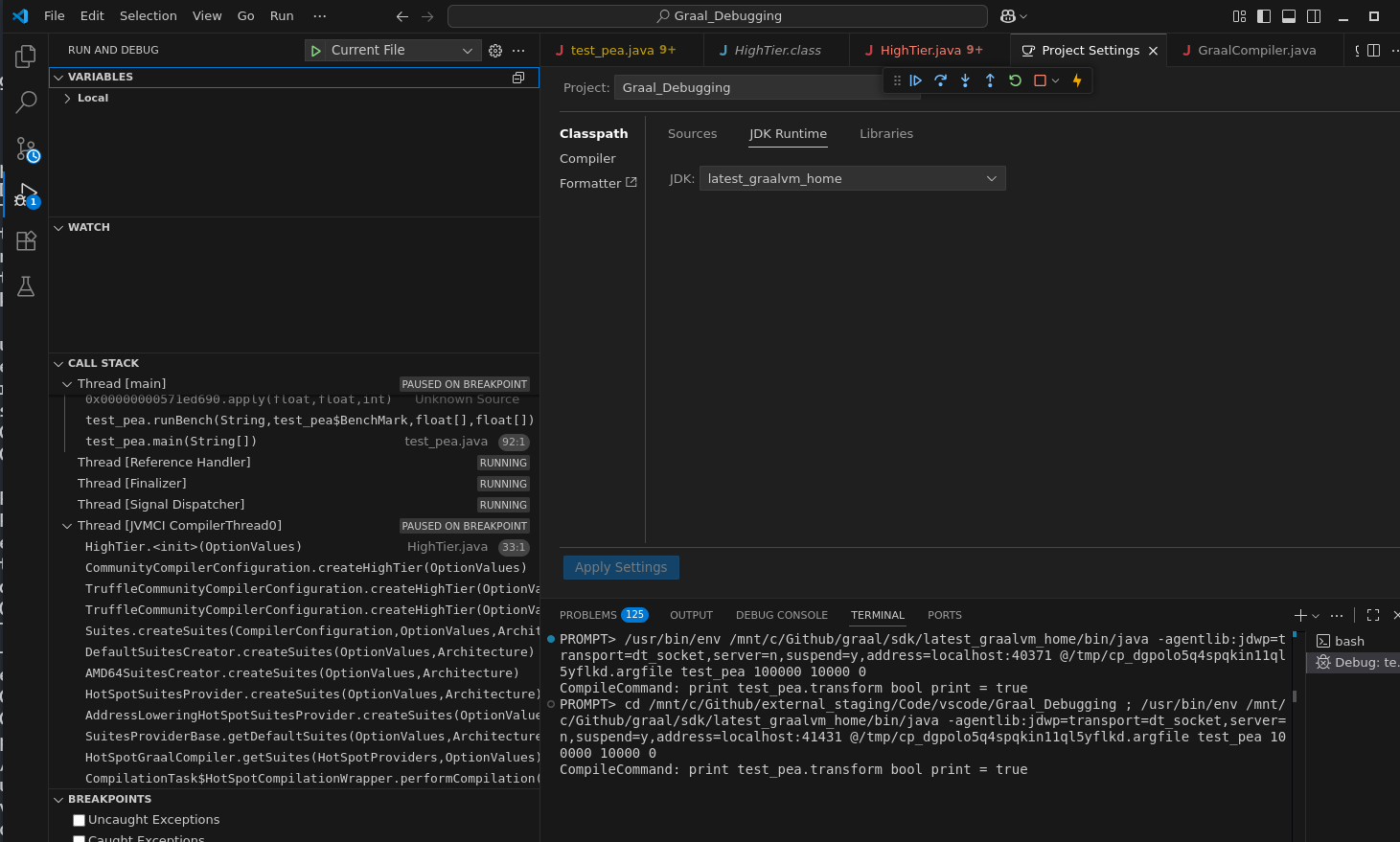
5/ export GRAAL\_HOME=<dir>/graal/sdk/latest\_graalvm\_home

Run - Graal compiler using JVMCI

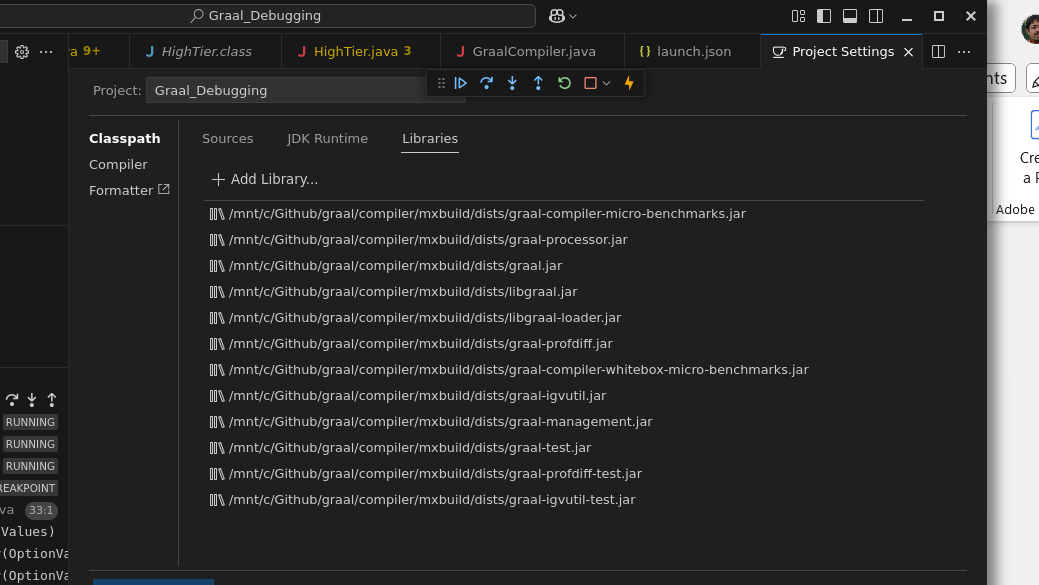
$GRAAL\_HOME/bin/java -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI --add-exports=java.base/jdk.internal.misc=jdk.graal.compiler -XX:-JVMCIPrintProperties -Djdk.graal.PrintPropertiesAll=false -Dgraalvm.locatorDisabled=true -XX:+UseJVMCICompiler -Djdk.graal.PrintCompilation=true -cp . test\_pea 100000 10000 0

1. **Debugging the Graal compiler**

Given that Graal itself is written in Java best way to debug it is by developing a Java test and then set the Java Runtime to the latest\_graalvm\_home directory under <dir>/graal/sdk/



Also import the \*graal\*.jar files



Add new launcher configuration

{

// Use IntelliSense to learn about possible attributes.

// Hover to view descriptions of existing attributes.

// For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387

"version": "0.2.0",

"configurations": [

{

"type": "java",

"name": "Current File",

"request": "launch",

"mainClass": "${file}"

},

{

"type": "java",

"name": "test\_pea",

"request": "launch",

"mainClass": "test\_pea",

"projectName": "Graal\_Debugging\_7e9848fa",

"vmArgs": "-server -XX:-TieredCompilation -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI --add-exports=java.base/jdk.internal.misc=jdk.graal.compiler -Djdk.graal.ShowDumpFiles=true -Djdk.graal.PrintGraph=Network -XX:+UseJVMCICompiler -XX:CompileCommand=Print,test\_pea::transform -XX:-TieredCompilation -Djdk.graal.PrintCompilation=false",

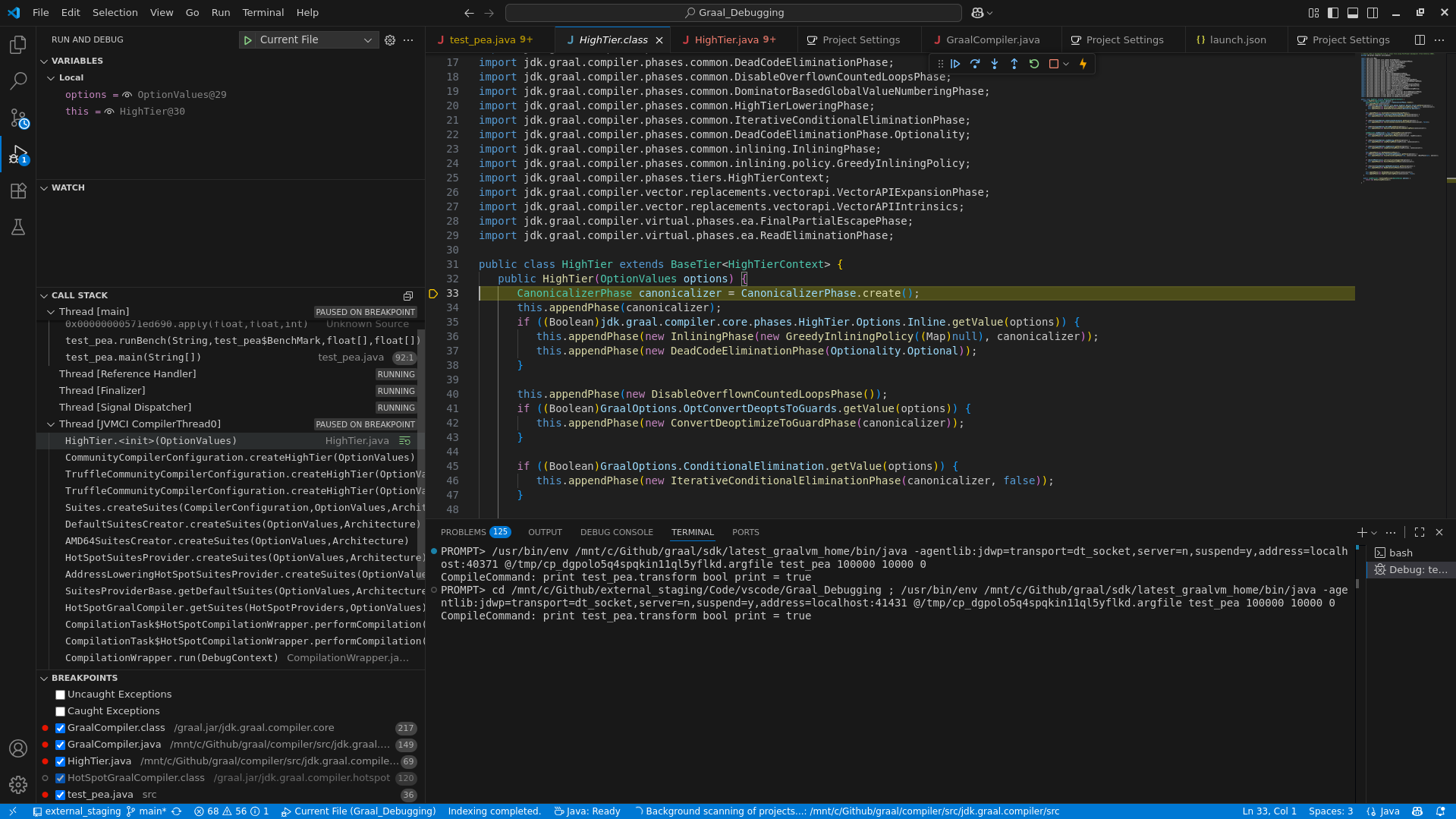
"args": "100000 10000 0"

}

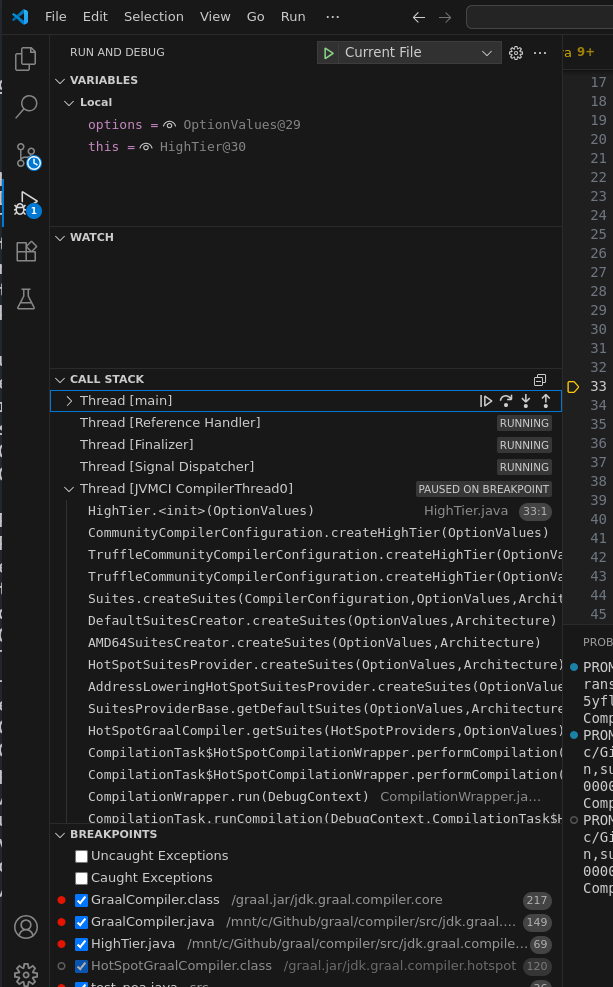
]

}

That’s it, launch the debugger after putting the breakpoint in the appropriate Graal compiler method:-

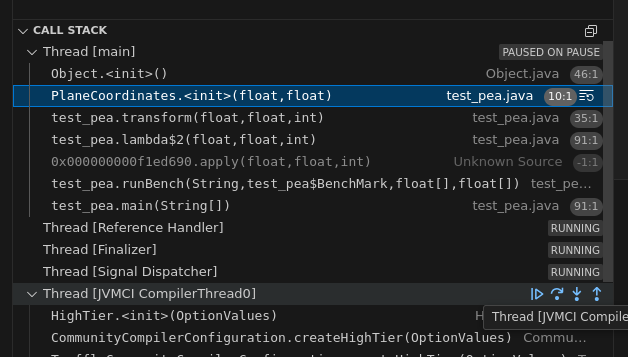


Check in the call stack, the new JVMCI Thread corresponds to the Graal compiler.



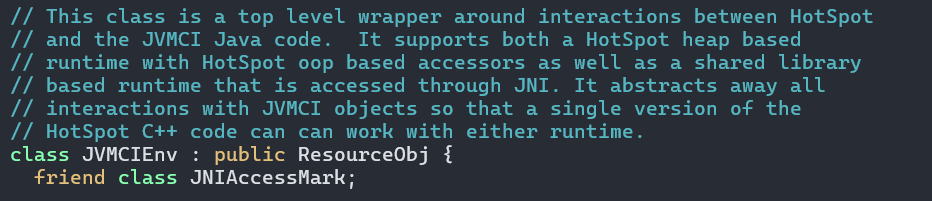
For hassle-free debugging, it's better to use -Xbatch, so that the compilation requests are blocking; otherwise, the interpreter may continue executing the code in the background.

Another alternative is to explicitly pause the execution of the application thread by clicking on the pause execution icon in the call stack



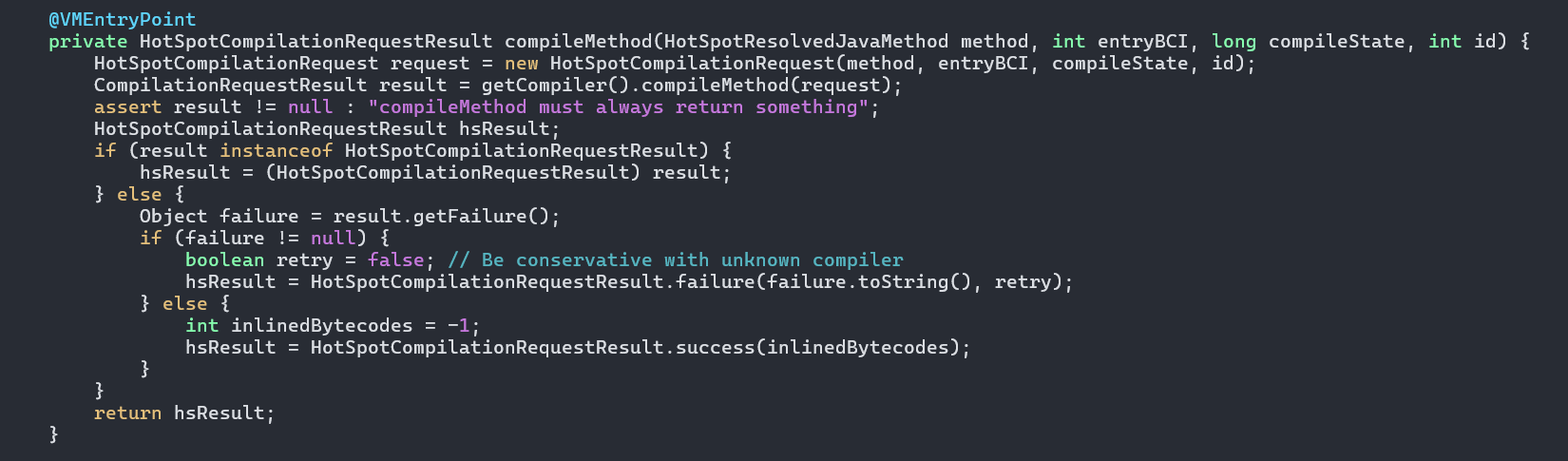
1. **Flow and important code pointers**

HotSpot VM (native ) -> JVMCI (Java interface to Graal compiler)

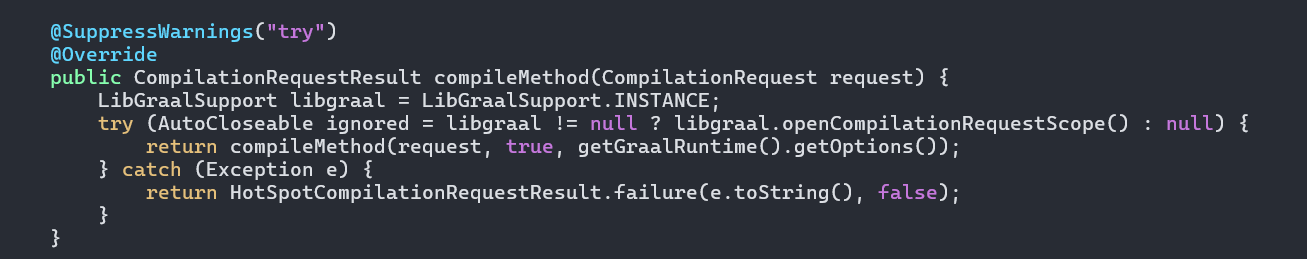


The Native Hotspot VM uses JavaCalls::call\_special to invoke a Java method directly, in this case JVMCI’s compileMethod, which generates a HotSpotCompilationRequest to be processed by the JVMCI compiler thread running the Graal compiler.

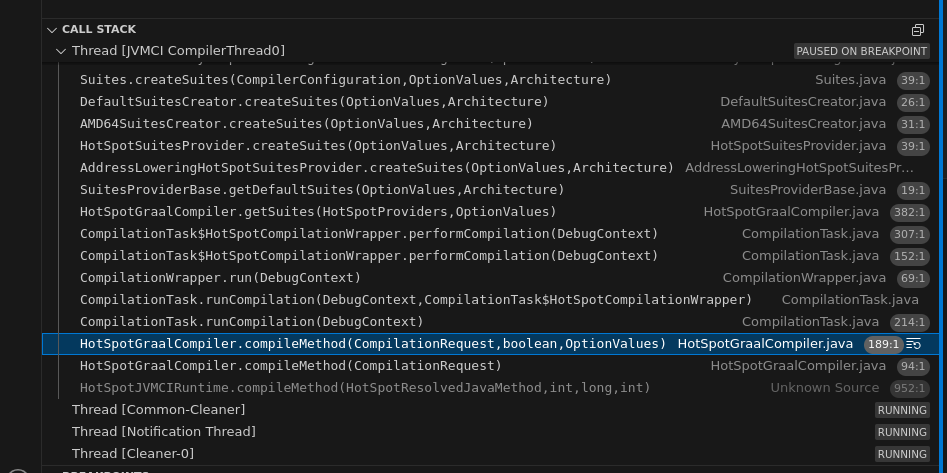
File: jdk/src/jdk.internal.vm.ci/share/classes/jdk/vm/ci/hotspot/HotSpotJVMCIRuntime.java



This compile request is received by HotSpotGraalCompiler thread

File: graal/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/hotspot/HotSpotGraalCompiler.java 

Here is the call stack of interest



The following file contains various compiler tuning options:-

*graal/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/core/common/GraalOptions.java*

All the options listed in above page can be invoked using -Djdk.graal.<OPTION>=<VALUE>

e.g.

**PROMPT>$GRAAL\_HOME/bin/java -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler -Djdk.graal.PrintCompilation=false -XX:+JVMCIPrintProperties -Djdk.graal.TargetVectorLowering=false -Djdk.graal.PrintPropertiesAll=true -cp . test\_pea 1000000 10000 0 | grep Vector**

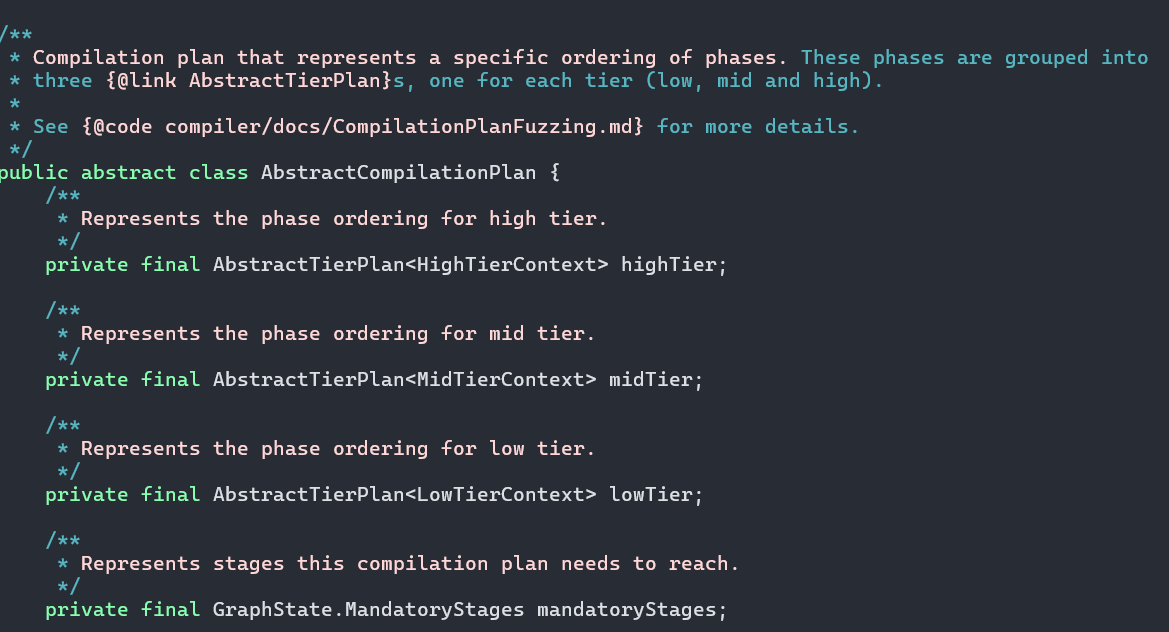
**jdk.graal.OptimizeVectorAPI = true [community edition] [Boolean]**

**Expand Vector API operations to optimized machine instructions**

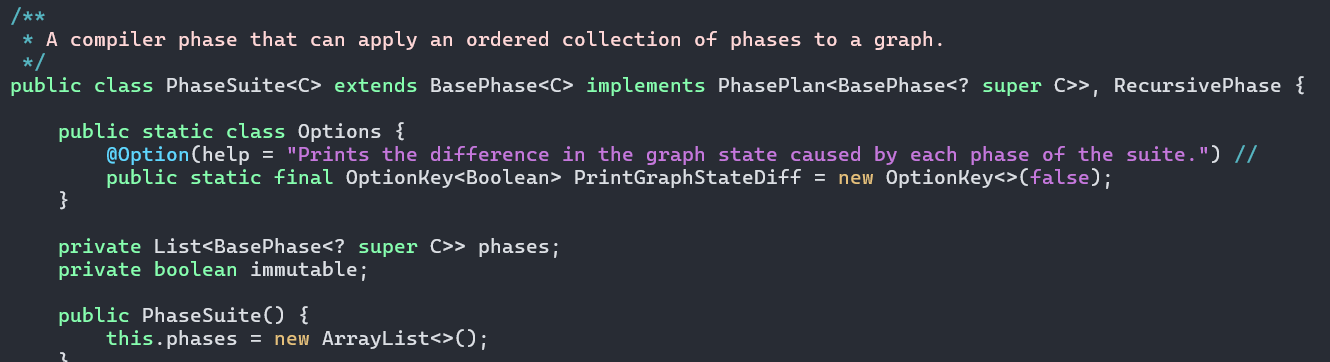
**jdk.graal.TargetVectorLowering := false [community edition] [Boolean]**

**PROMPT>**

**Phases are divided into multiple tiers: High, Mid, and Low. Phase ordering within a respective tier is captured by AbstractCompilationPlan**

****

**Phases are organized into multiple Suites**

****

**Abstract BasePhase has an instance-specific method run, which is overridden by different concrete optimization phases. This method receives a StructuredGraph and Context as parameters.**

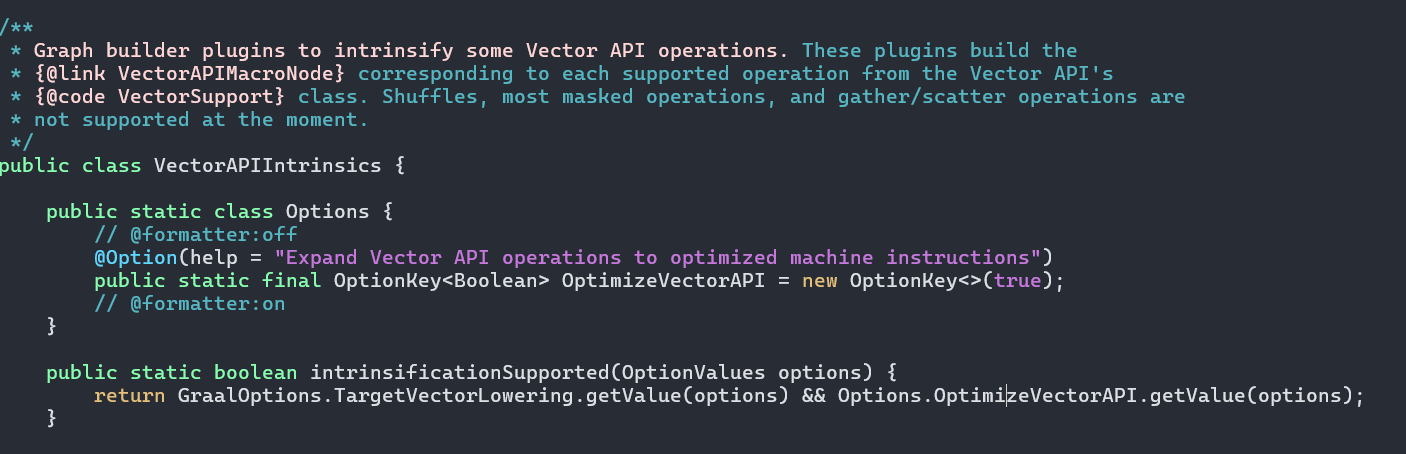
**protected abstract void run(StructuredGraph graph, C context);**

**In the context of VectorAPI, AMD64VectorLoweringPhase is of interest.**

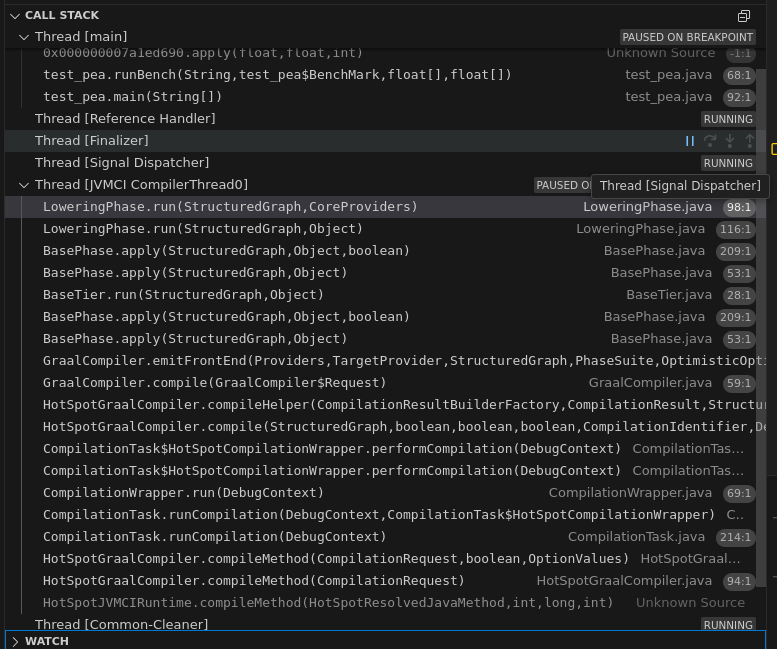
**Another interesting option in context of VectorAPI is**

**jdk.graal.OptimizeVectorAPI = true [community edition] [Boolean]**

**Expand Vector API operations to optimized machine instructions**

****

**Here is an execution stack of method compilation.**

****



A screenshot of a computer program

Description automatically generated

Front-end HIR, High, Mid and Low Tier compiler phases shared b/w JIT and AOT:-

Different backends implement the LIRGenerationProvider interface.

1. Substrate VM AOT Backend: SubstrateAMD64Backend
2. Hotspot JIT Top Tier Compiler Backend: AMD64HotSpotBackend

A screenshot of a computer program

Description automatically generated

emitFrontEnd()

* Front-end -> GraphBuilderPhase -> createBytecodeParser -> HotSpotBytecodeParser ->

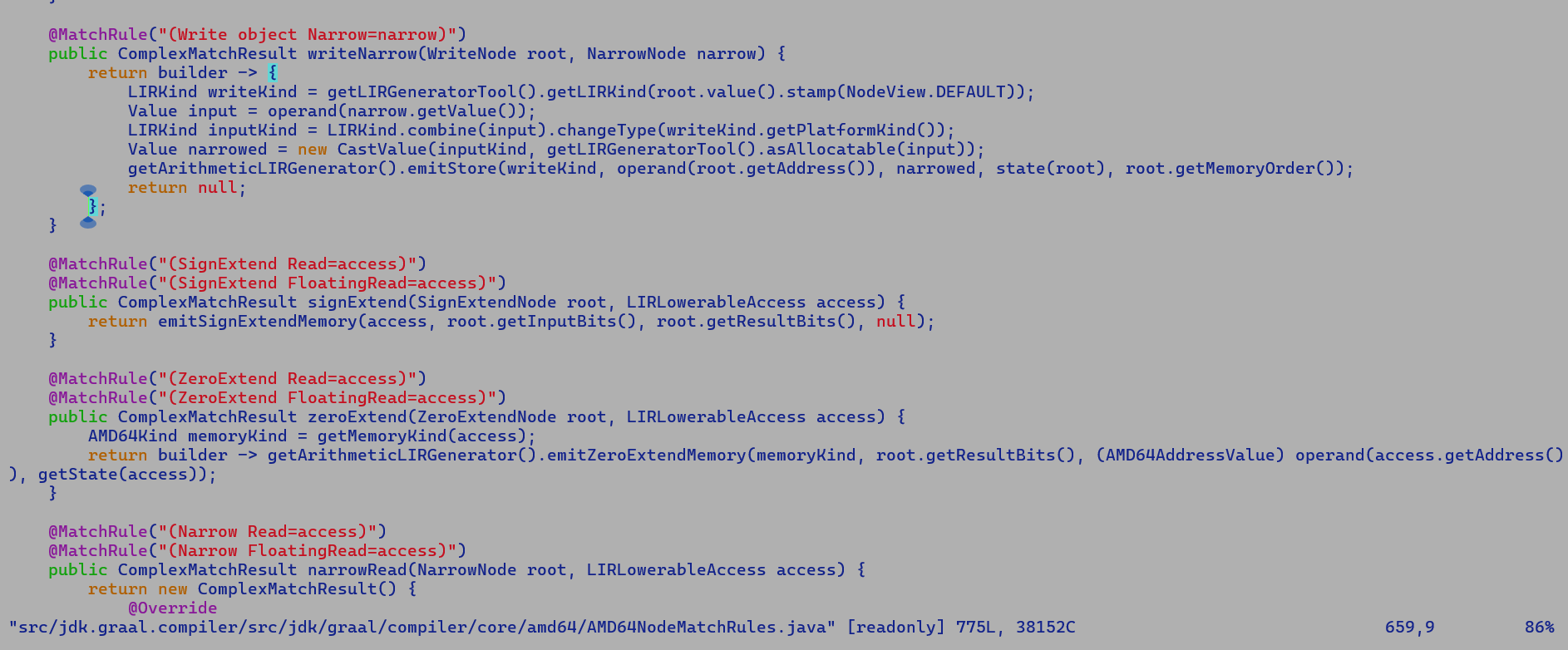
A screenshot of a computer program

AI-generated content may be incorrect.

* + processBytecode
* StructuredGraph
* Optimization MidTier.

emitBackend()

* LIG generation – Matching

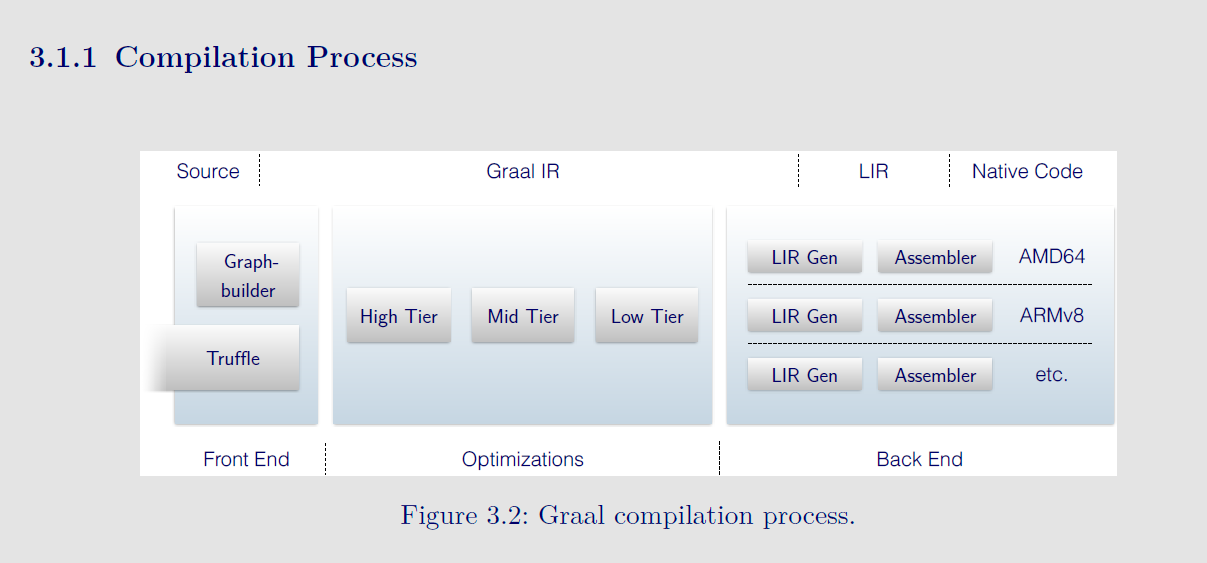


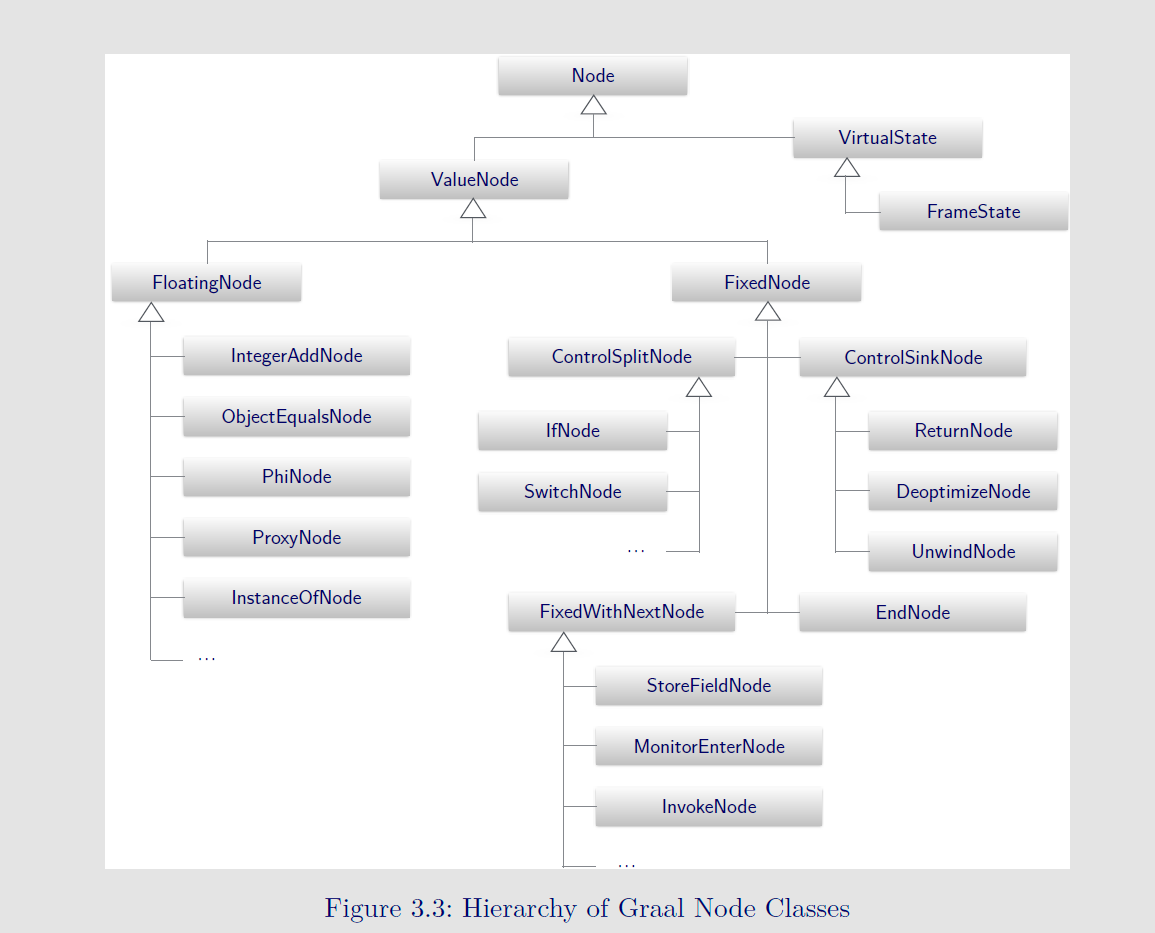
* createJVMCIBackend()



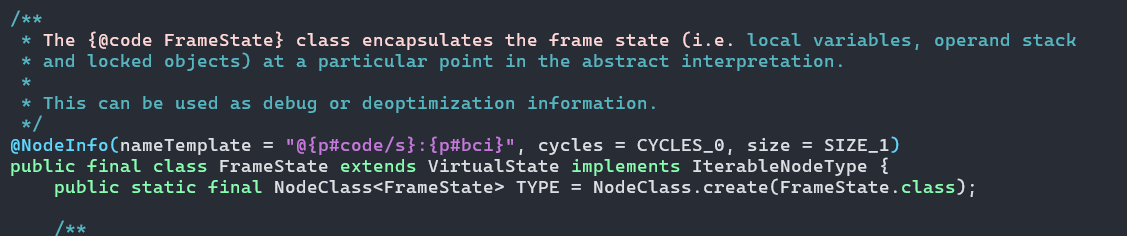
1. **Exploring Graal’s partial escape analysis**

<https://bugs.openjdk.org/browse/JDK-8366137>



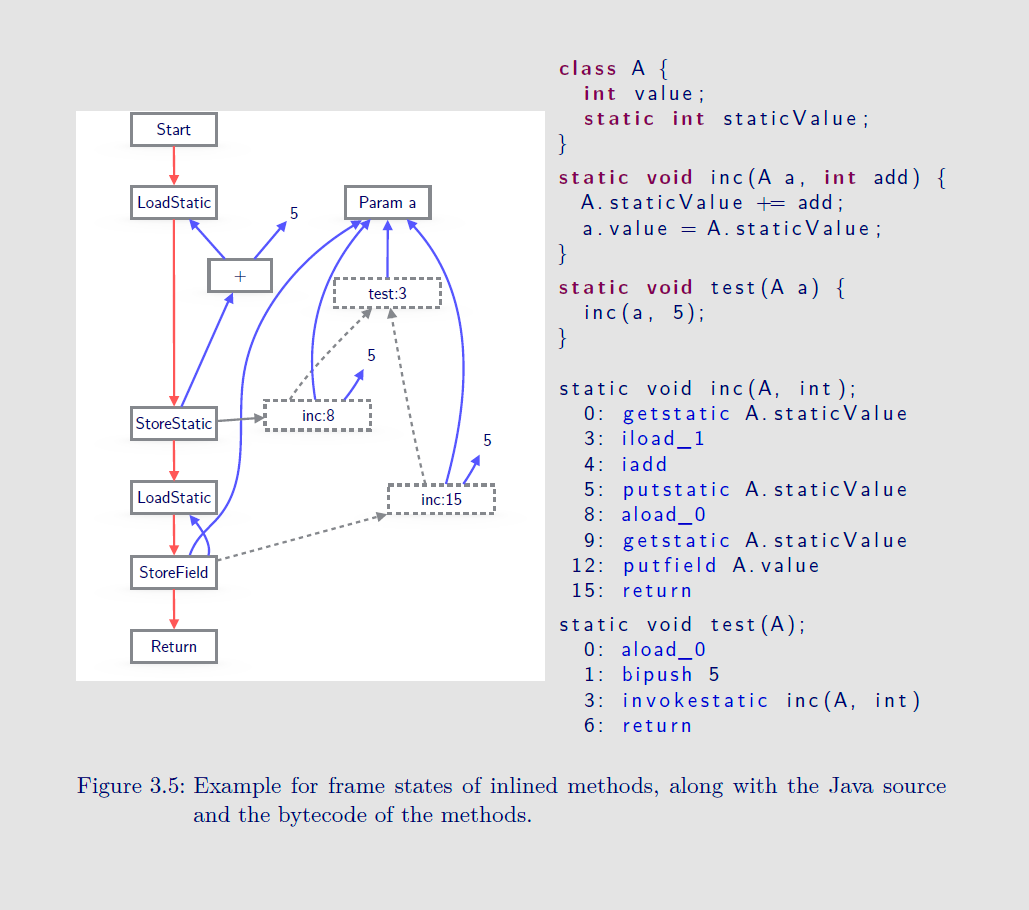


FrameState in Graal compiler is analogous to JVMState in C2



JVMState is associated with various SafePointNodes, around loop back, method return, and at the call site. build\_start\_state creates initial state of method creation by ParserGenerator. A chain of JVMState is formed if a node is inlined. During code emission, the compiler dumps the stack of frames associated with Safepoints into the debug stream. Multiple VM clients, such as deoptimization, use this to reconstruct the interpreter state. It is also used during root set enumeration, along with OopMaps, to find the location of oops.

Runtime always expects a valid OopMap against each frame; auxiliary structure PCDescriptor maintains this mapping.



[C2 tip] Useful iterators:-

* OopMapStream
* vframeStream
  + vframeStreamCommon