Virtual Machines and Runtime

A focus into the Java and Javascript Runtime Environments

Table of Contents

What is a Virtual Machine?

- ➔ A virtual machine is a software-defined computer whose purpose is to simulate a physical machine, running on a « host » machine as a « guest ».
- ➔ The **virtual machine** is sandboxed from the rest of the system.
- ➔ Hardware is simulated: CPU, memory, hard drive, network interfaces, *etc*.

Types of Virtual Machines

SYSTEM VIRTUAL MACHINE

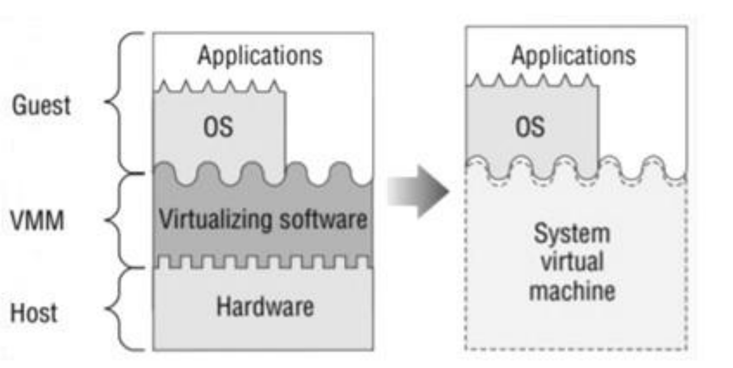
- ➔ Simulate a physical machine
- ➔ Sharing of a host computer's physical resources between multiple VMs, running its own copy of the OS
- ➔ Using an hypervisor (software that creates and runs VMs) that can run directly on hardware (VMware ESXI) or on top of an OS (VirtualBox).

PROCESS VIRTUAL MACHINE

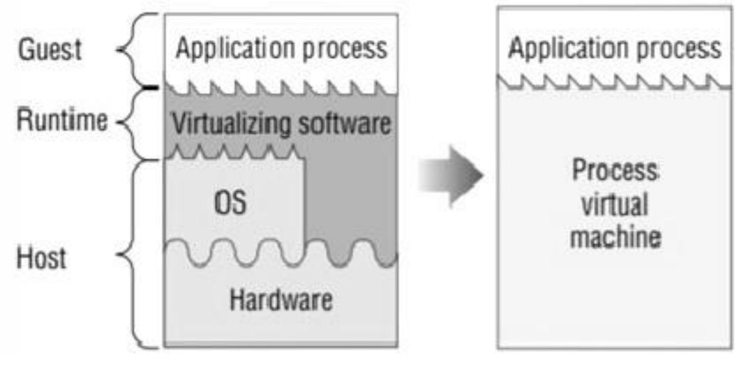
- ➔ Used for executing a single process
- ➔ Masks the information of the underlying hardware/OS
- ➔ Ex: Java Virtual Machine

Types of Virtual Machines

SYSTEM VIRTUAL MACHINE



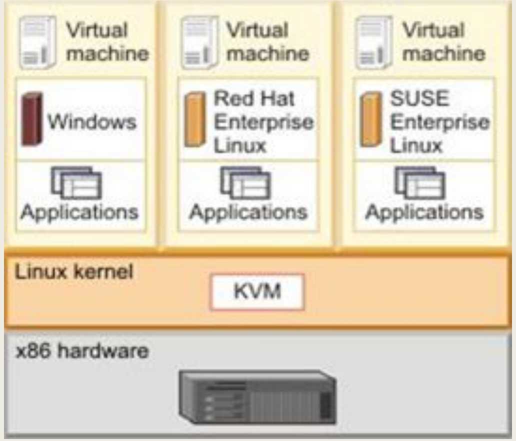
PROCESS VIRTUAL MACHINE



Hypervisors

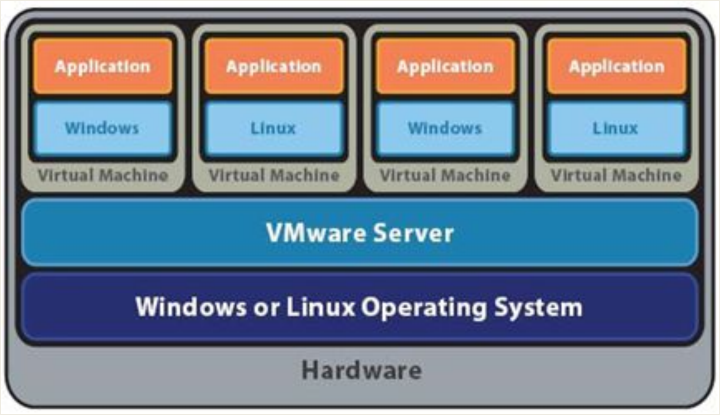
TYPE I HYPERVISOR

➔ Runs directly on the hardware.



TYPE II HYPERVISOR

➔ Runs on top of an OS.



Container VS Virtual Machine

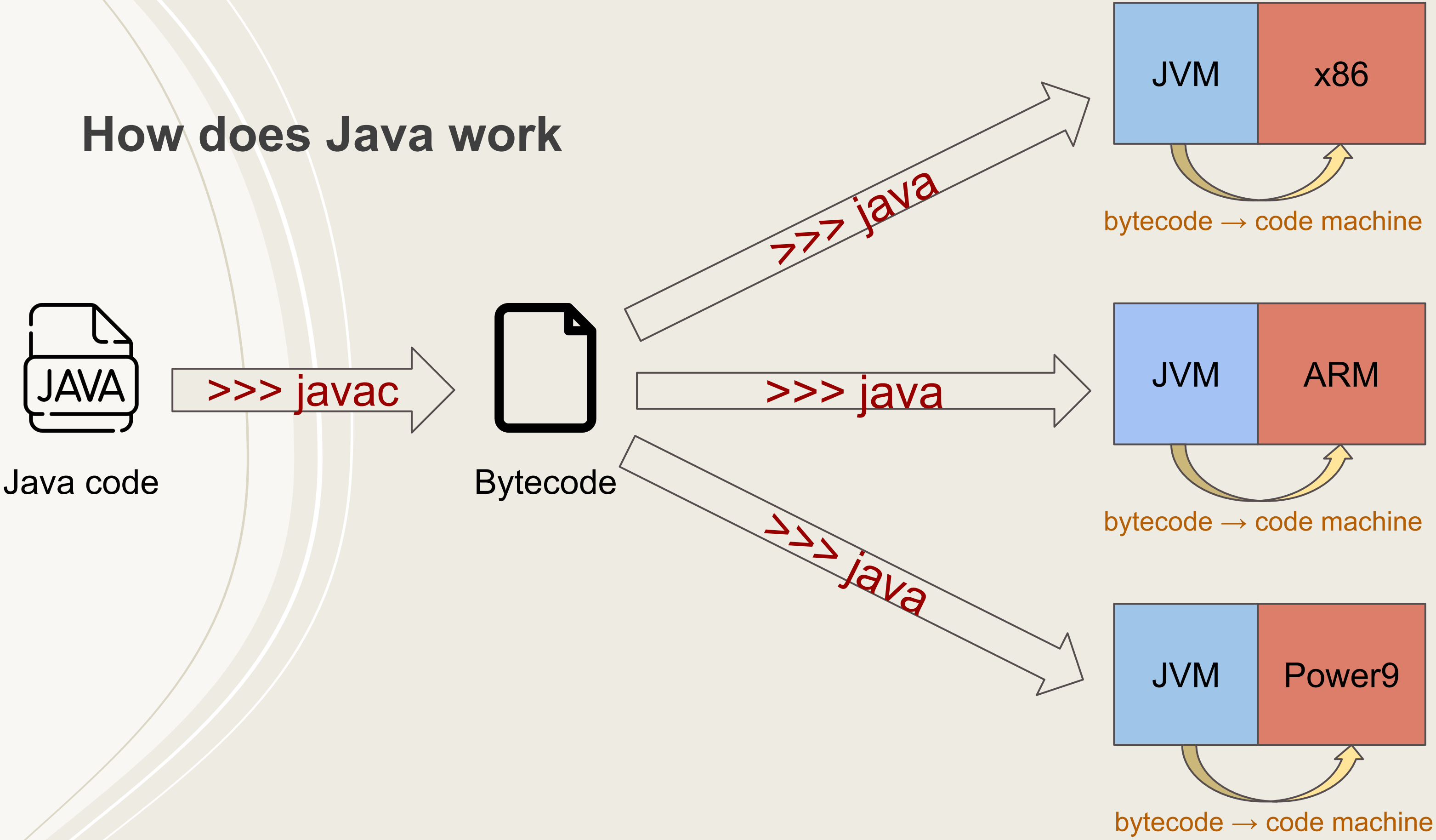
Both are running an isolated application on a single platform.

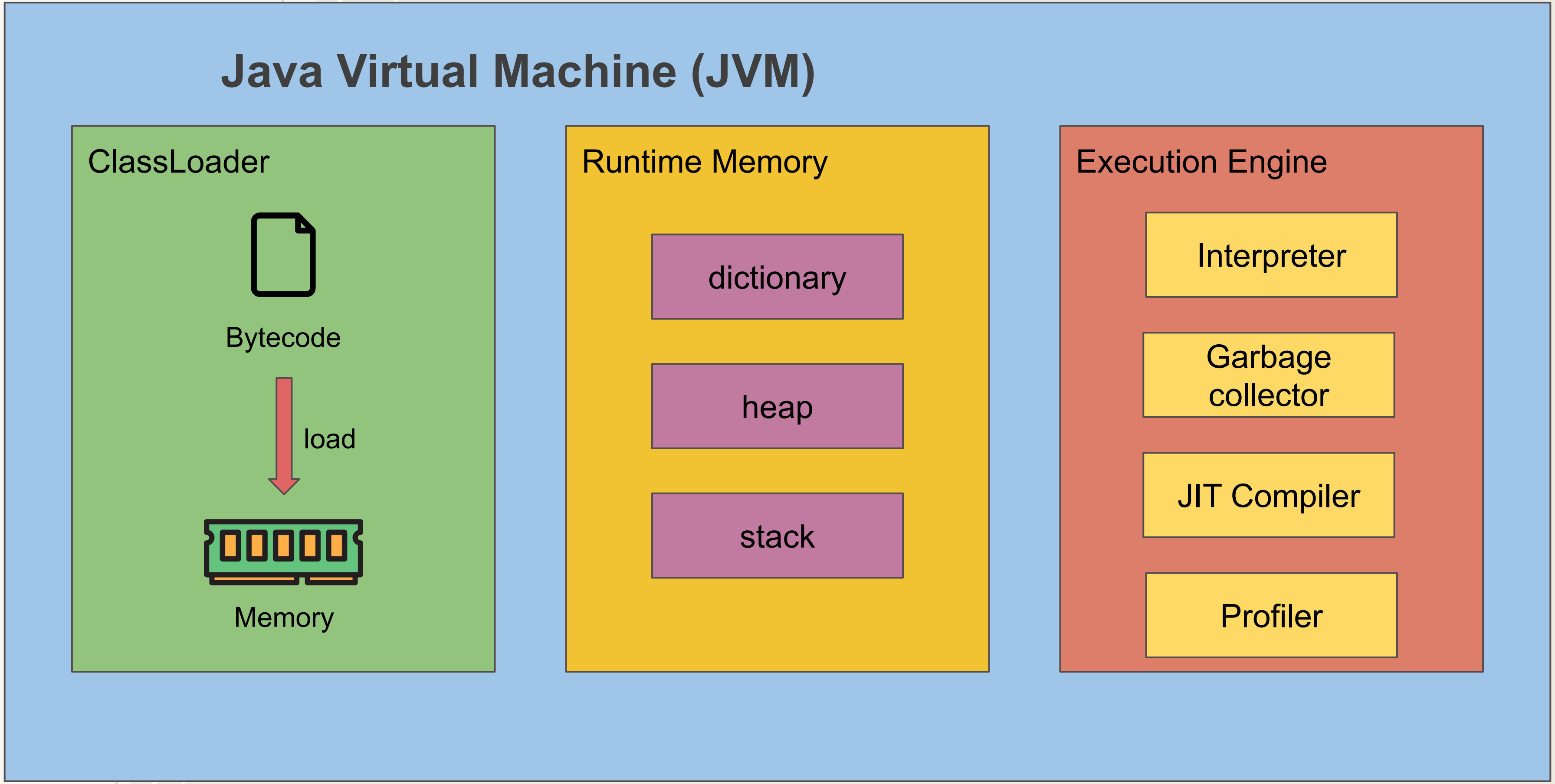
CONTAINER	VIRTUAL MACHINE
Package a single application along with its dependencies	Virtualize the hardware layer to create a “computer”
Provide shared OS services from the underlying host	Managed by a hypervisor
Less overhead	Larger and slower to boot
Containers that are on the same host share the same OS kernel	Isolated from one another, with a completely separate OS

Java Virtual Machine (JVM)

- ➔ Virtual Machine that allows the execution of programs compiled to Java bytecode.
- ➔ Converts Java bytecode into machine language and execute it.
- ➔ Follows a standard required in a JVM implementation.
- ➔ The JVM simulates a 32-bit machine.
- ➔ Offers a garbage collector for removing unreferenced objects.
- ➔ JVM is “platform dependant” : JVMs are available for many hardware and software platforms.
- ➔ JVM gives Java the flexibility of platform independence

How does Java work




Java Virtual Machine (JVM)
ClassLoader
Bytecode
load
Memory
Runtime Memory
dictionary
heap
stack
Execution Engine
Interpreter
Garbage collector
JIT Compiler
Profiler

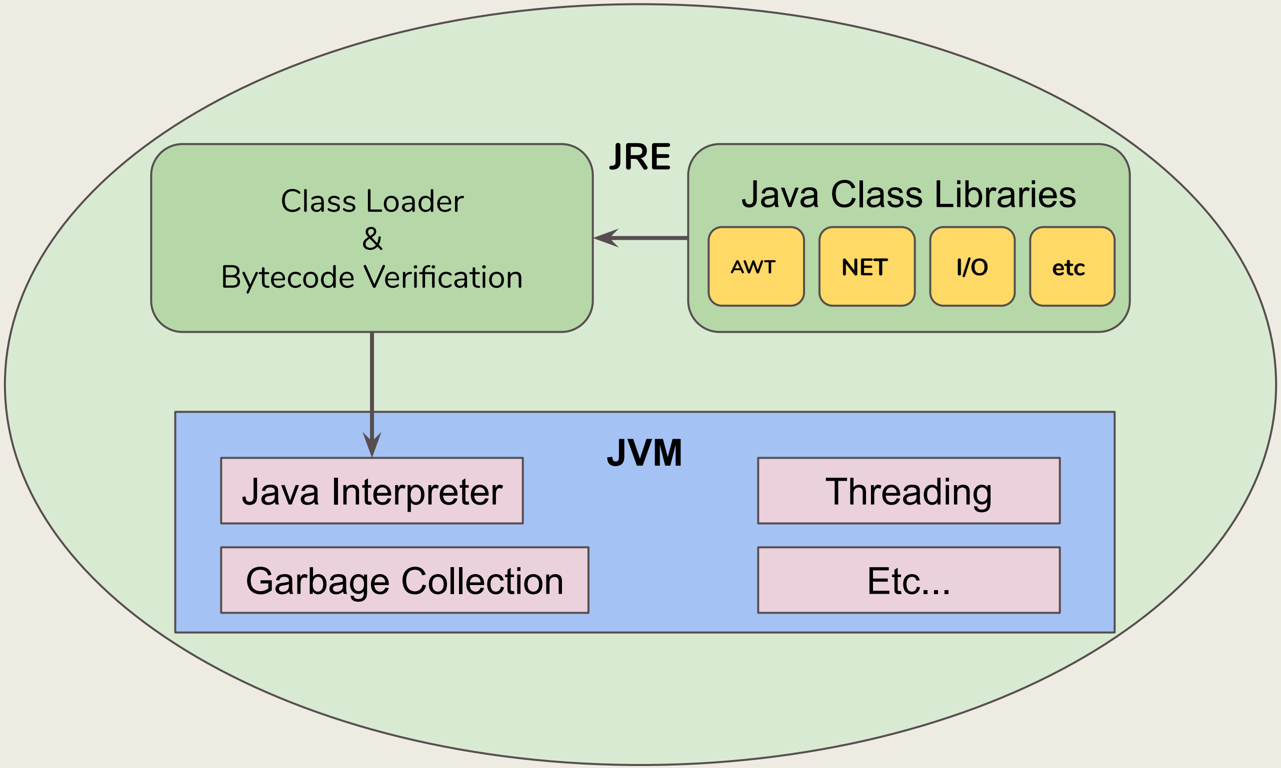
Runtime Environment (RTE)

- ➔ The runtime of a program is the execution and running phase.
- ➔ A Runtime Environment provides all the functionalities and dependencies necessary to run a program independently of the underlying operating system, which allows the program to have the same user interfaces regardless of the OS.
- ➔ Provides basic functions for memory, network and hardware.

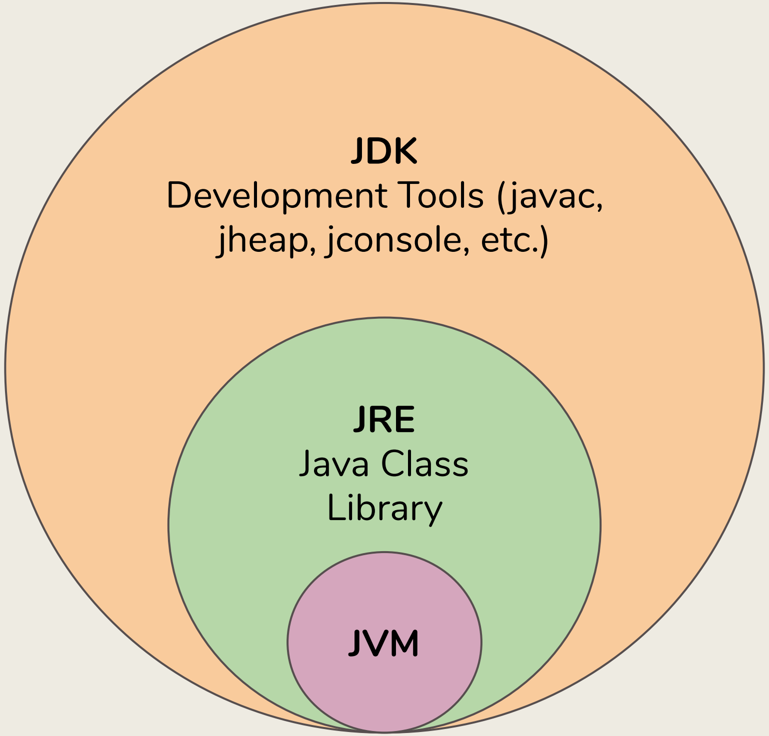
Java Runtime Environment (JRE)

- ➔ Combines the Java code created by the JDK with the dependencies required (libraries) and creates an instance of the JVM to run the resulting program.
- ➔ Enables a Java program to run in any OS without modification.
- ➔ Automatic memory management via Garbage Collection

Java Runtime Environment (JRE)



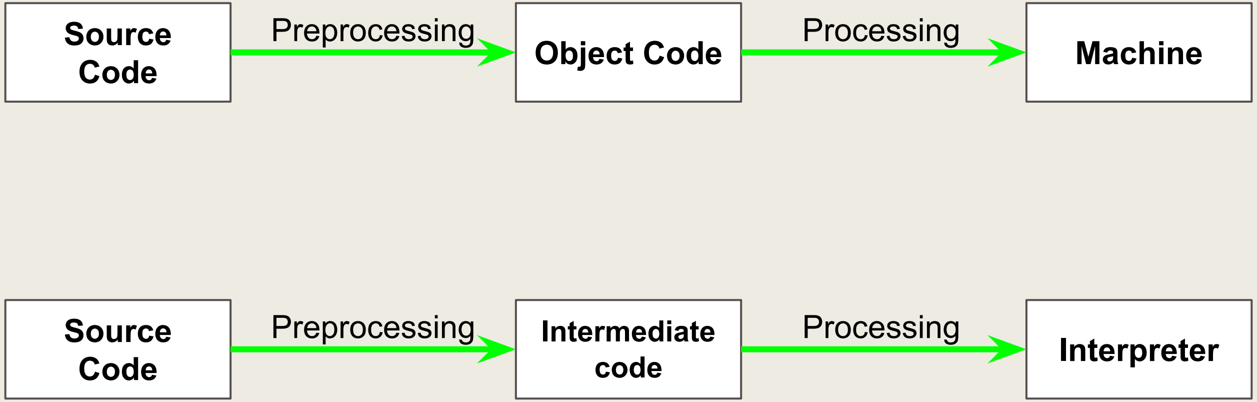
The JVM, the JRE and the JDK



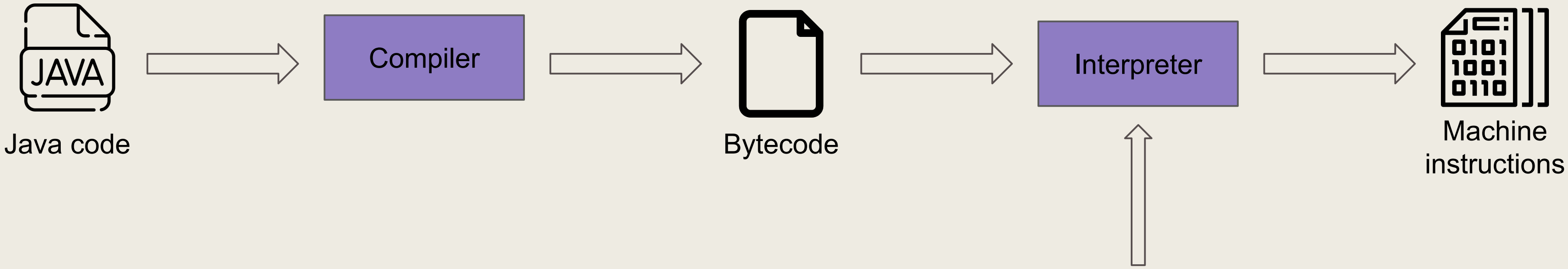
Interpreter vs Compiler

INTERPRETER	COMPILER
Translates one statement at a time into machine code	Scans the entire program and translates it into machine code at once
Takes less time to analyse the source code, but the process execution time is much slower	Takes a lot of time to analyse the source code, but the process execution time is much faster
Does not generate intermediary code, highly efficient in terms of memory	Always generates intermediary object code. Needs further linking, so more memory is needed
Keeps translating the program until an error is found. Execution is stopped if an error is spotted	Generates error messages after the whole program has been scanned
Languages using interpreters : Ruby or Python, JavaScript	Languages using compilers : C, C++, Java

Interpreter vs Compiler



Interpreter

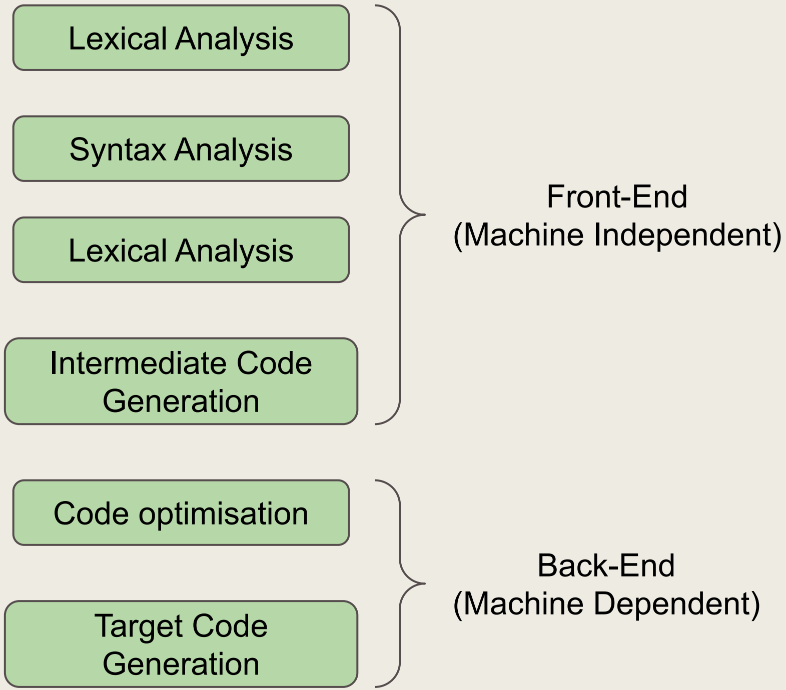


bytecode instruction	machine instruction
line 1	line 1
...	...

Compiler

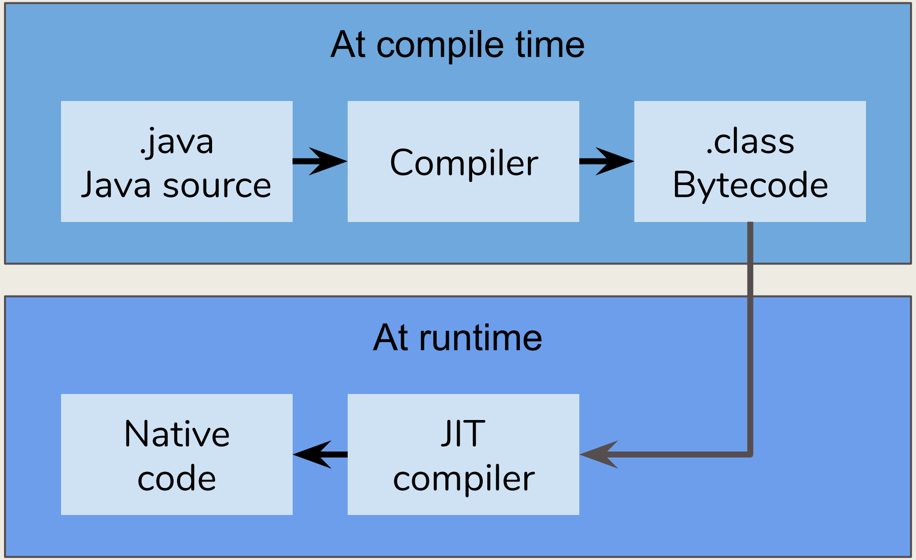
- ➔ A compiler is a program that translates source code into another lower-level language.
- ➔ Its role is to check for all possible errors in a source program, such as spelling errors, variables, types.
- ➔ Il compile tout le code source en une fois et s’il n’y a aucune erreur, on obtient un exécutable.
- ➔ Different types of compilers : we gonna talk about the Java JIT compiler.

Phases of a compiler



Just-In-Time (JIT) Compiler

- Used to improve the performance of java applications at run time.
- Compiles « just in time » bytecodes into native machine code at runtime.
- bytecodes reformulated in a representation called ***trees***
- The JVM calls the compiled code instead of interpreting it.
- JIT compilers combine interpreter and compiler principals to get the best of both worlds.



Phases of JIT Compilation

Inlining	Local optimizations	control flow optimizations	global optimizations	native code generation
Trees of smaller methods are merged or “inlined” into the trees of their callers	Local optimizations analyze and improve a small section of the code at a time	Analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency	Work on the entire method at once. Requiring larger amounts of compilation time, but can provide a great increase in performance.	The trees of a method are translated into machine code instructions

JIT : Multiple strategies

- ➔ Just in time : transform into assembly at the first call
- ➔ Tiered - 2 JITs : 1 fast and simple and 1 optimized (slower)
- ➔ Mixed-mode : Interpreter + 1 JIT or more
- ➔ Ahead of time : code is optimized at the installation

Advantages / Disadvantages of a JIT compiler

ADVANTAGES	DISAVANTAGES
Less memory usage	Startup time can be slow
Run after a program starts	Heavy usage of cache memory
Code optimization is done while the code is running	Can increase the level of complexity of a program
Can utilize different levels of optimization	

JIT optimizations on loops

- ➔ Loop reduction and inversion
- ➔ Loop striding and loop-invariant code motion
- ➔ Loop unrolling and peeling
- ➔ Loop versioning and specialization

Optimization: Loop inversion

➔ Transforms a while loop to an if block containing a do while loop to eliminate the jumps.

```
void foo(int n) {
  while (n < 10) {
    use(n);
    ++n;
  }
  done();
}
```

```
void foo(int n) {
  if (n < 10) {
    do {
      use(n);
      ++n;
    }
    while (n < 10);
  }
  done();
}
```

Optimization: Loop unrolling

➔ The JIT compiler opens up the loop and repeats the corresponding Assembly instructions one after another.

```
private static double[] loopUnrolling(double[][] matrix1,
double[] vector1) {
	double[] result = new double[vector1.length];

	for (int i = 0; i < matrix1.length; i++) {
	for (int j = 0; j < vector1.length; j++) {
			result[i] += matrix1[i][j] * vector1[j];
		}
	}

	return result;
}
```

```
private static double[] loopUnrolling2(double[][] matrix1, double[] vector1) {
	double[] result = new double[vector1.length];

	for (int i = 0; i < matrix1.length; i++) {
		result[i] += matrix1[i][0] * vector1[0];
		result[i] += matrix1[i][1] * vector1[1];
		result[i] += matrix1[i][2] * vector1[2];
		// and maybe it will expand even further - e.g. 4 iterations
		// adding code to fix the indexing
	}

	return result;
```

JIT optimizations on Method Calls

➔ When possible, the JIT compiler would try to inline method calls and eliminate the jumps of going there and back, the need to send arguments, and returning a value and transferring its whole content to the calling method.

```
private static void calcLine(int a, int b, int from, int to) {
        Line l = new Line(a, b);
        for (int x = from; x <= to; x++) {
                int y = l.getY(x);
                System.err.println("(" + x + ", " + y + ")");
        }
}
static class Line {
        public final int a;
        public final int b;
        public Line(int a, int b) {
                this.a = a;
                this.b = b;
        }
        // Inlining
        public int getY(int x) {
                return (a * x + b);
        }
}
```

```
private static void calcLine(int a, int b, int from, int to) {
        Line l = new Line(a, b);
        for (int x = from; x <= to; x++) {
                int y = (l.a * x + l.b);
                System.err.println("(" + x + ", " + y + ")");
        }
}
```

➔ Eliminate the jump, the send of the arguments l and x, and the returning of y

JIT Optimizations – Null Check Elimination

- Checking for null is common in Source Code
- But these checks can be eliminated from optimized Assembly Code
  - To avoid unnecessary jumps that slow down execution
- At runtime, the JIT knows if an object is null or not
- So it can perform this kind of transformation :

```java
private static void runSomeAlgorithm(Graph graph) {
        if (graph == null) {
                return;
        }

        // do something with graph
}
```

```java
private static void runSomeAlgorithm(Graph graph) {

        // do something with graph
}
```

- “Uncommon trap” mechanism if the condition eventually happens:
  - JIT deoptimize code

JIT Optimization – Branch Prediction

- Similar to Null Check Elimination
- Try to decide which lines of code are "hotter" (*ie. happen more often*)
- Branches of an IF condition can be reordered to reduce the number of jumps in Assembly Code.

```java
private static int isOpt(int x, int y) {
        int veryHardCalculation = 0;

        if (x >= y) {
                veryHardCalculation = x * 1000 + y;
        } else {
                veryHardCalculation = y * 1000 + x;
        }
        return veryHardCalculation;
}
```

```java
private static int isOpt(int x, int y) {
        int veryHardCalculation = 0;

        if (x < y) {
                // this would not require a jump
                veryHardCalculation = y * 1000 + x;
                return veryHardCalculation;
        } else {
                veryHardCalculation = x * 1000 + y;
                return veryHardCalculation;
        }
}
```

Example assuming that, in most cases, x < y

Benchmarks performed

- ➔ Benchmark performed on two big arrays:
  - ◆ Array of native type (int)
  - ◆ Array of two different objects (boxed ints)

- ➔ Focus on the loop iteration performance by going through all elements of the arrays.

- ➔ Focus on the access to a variable boxed/non boxed by a class.