Improve your apps performance by understanding how JVM works

May 2023

CIKLUM

CIKLUM

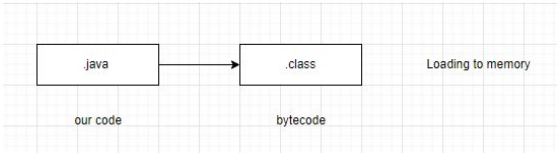
Java Community in Ciklum

CIKLUM

Contents

CIKLUM

* Compiled vs Interpreted



COMPILATION

Translate high-level code into machine code (or bytecode in our case)
- Lexical analysis (tokenization, keywords, identifiers, literals, operators)
- Syntax analysis (parse the code)
- Semantic analysis (types, dependencies, scopes)
- Intermediate code generation & Optimization
- Code generation

Faster execution than interpreter
Platform dependent

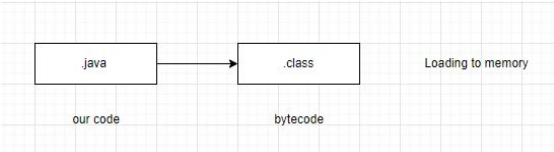
It is executed by
- machine's processor
- interpreter
- JIT + interpreter

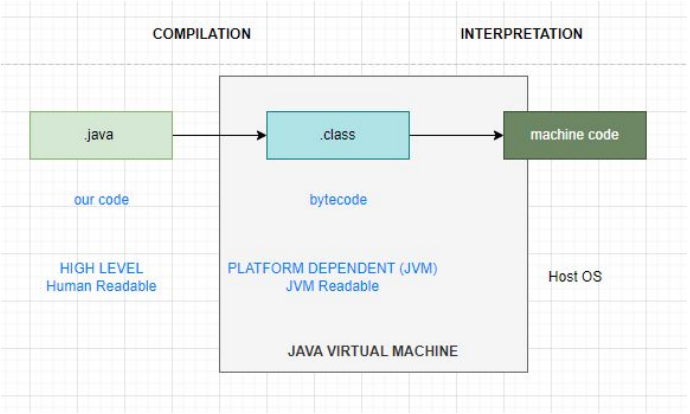
INTERPRETATION

Code is directly executed line by line, sequentially in real time
- Source Code interpretation
- Dynamic execution (if code is changed at runtime)
-

Slower execution than compiled
Platform independent

* Compiled vs Interpreted





COMPILATION

Translate high-level code into machine code (or bytecode in our case)
- Lexical analysis (tokenization, keywords, identifiers, literals, operators)
- Syntax analysis (parse the code)
- Semantic analysis (types, dependencies, scopes)
- Intermediate code generation & Optimization
- Code generation

Faster execution than interpreter
Platform dependent

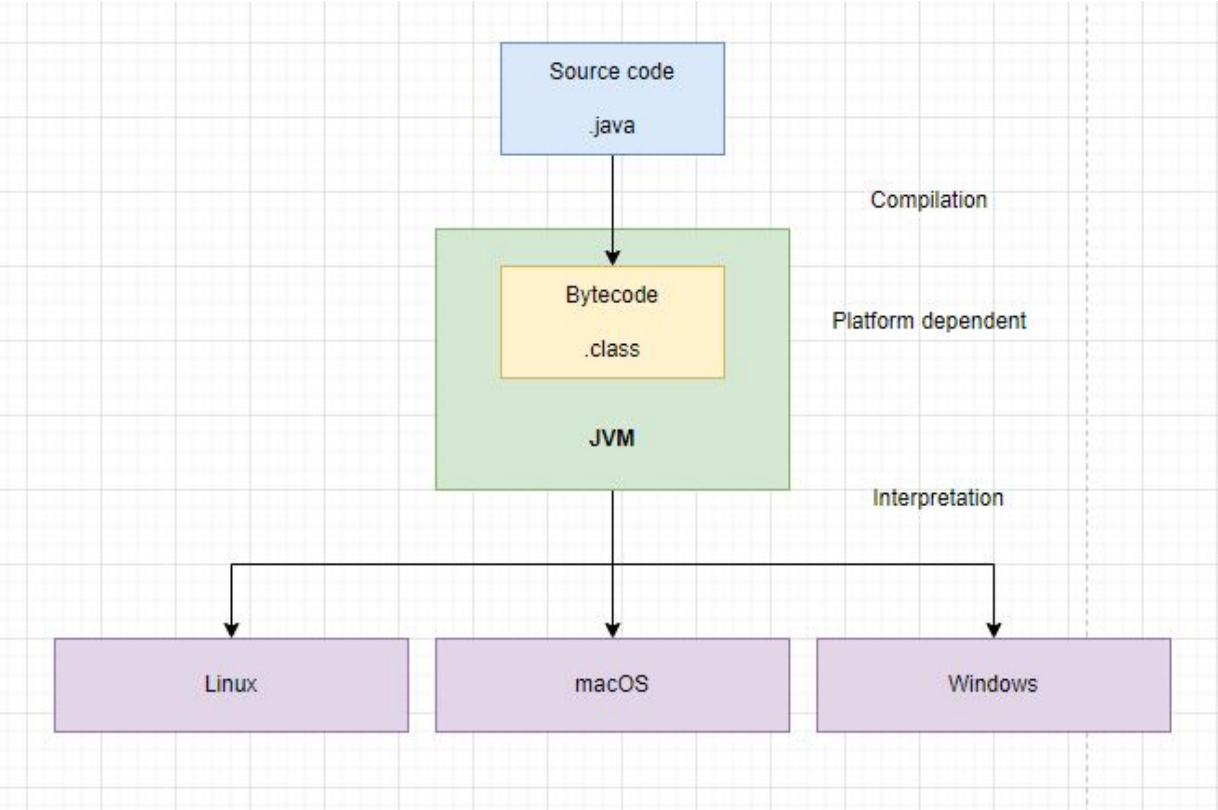
It is executed by
- machine's processor
- interpreter
- JIT + interpreter

INTERPRETATION

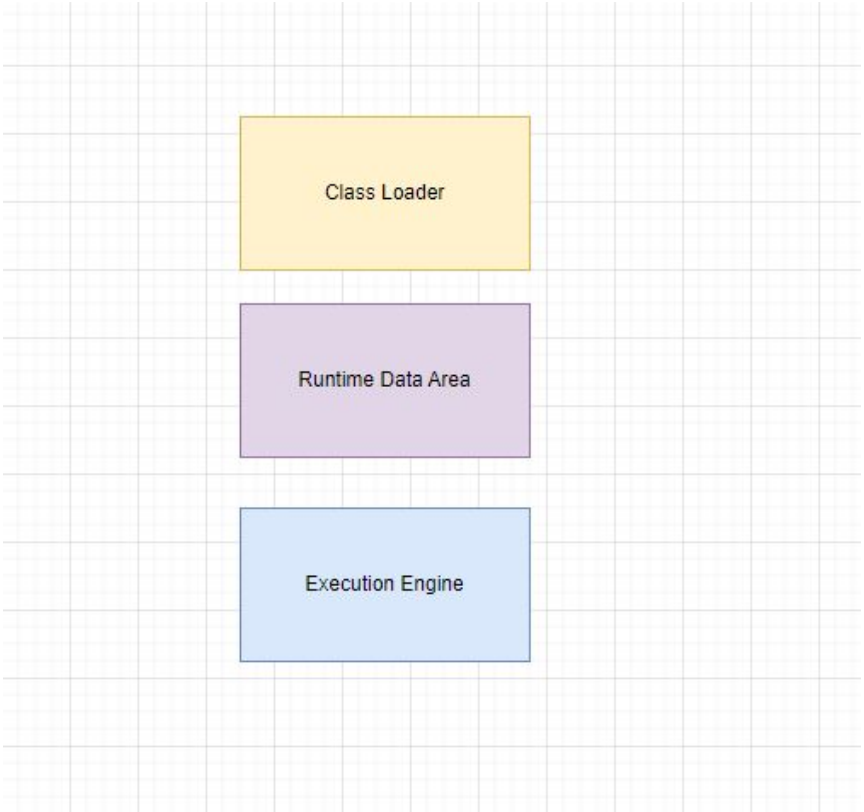
Code is directly executed line by line, sequentially in real time
- Source Code interpretation
- Dynamic execution (if code is changed at runtime)
-

Slower execution than compiled
Platform independent

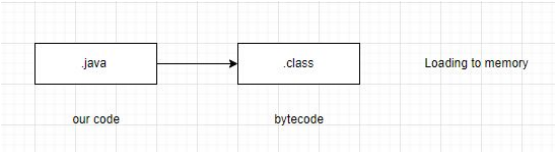
JVM Architecture

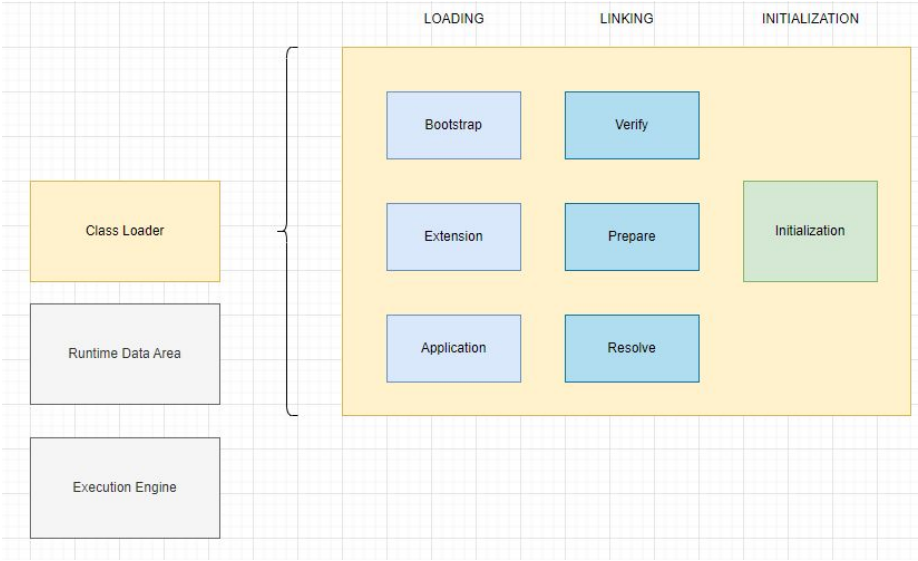


JVM Architecture



- Class Loader





LOADING

Bootstrap -> RT.jar

```
System.out.println("hello");
```

Extension -> $JAVA_HOME/jre/lib/ext OJDBC.jar

Application -> Main.java Main.class

LINKING

Verify -> check loaded classes

Prepare -> static variables mem allocation

Resolve -> Replace references

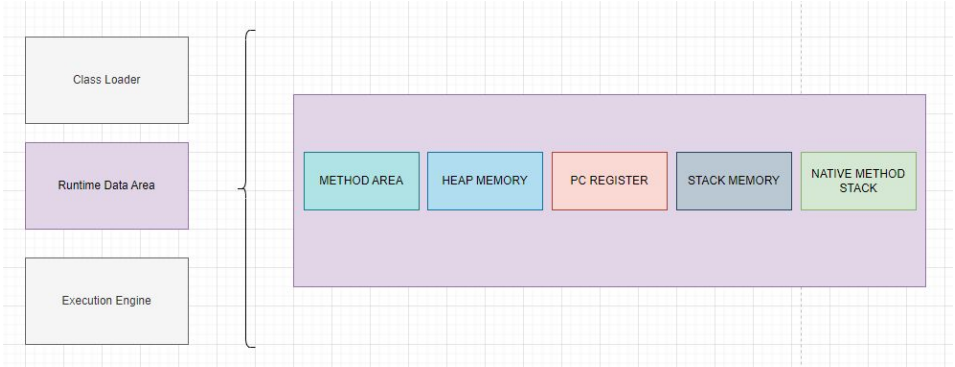
```
MyClass myClass = new MyClass();
myClass.myMethod( str: "hello");
```

INITIALIZATION

All static variables are assigned
Executes static initializers

```
static int myStaticVariable;

static {
    // Initialization code
    myStaticVariable = 42;
    System.out.println("Static initializer executed");
}
```

- Runtime data area



METHOD AREA -> Class data

```
Class MyClass{}
```

```java
public class MyClass {

    static int myStaticVariable = 1;

    public void myMethod(String str) {
        System.out.println(str);
    }
```

HEAP MEMORY -> Objects and instance variables and default initializations

```
new MyClass()
```

```java
@Component
public class MyClass {

    private int myInstanceVariable;

    public int getMyInstanceVariable() {
        return myInstanceVariable;
    }

    public void setMyInstanceVariable(int value) {
        myInstanceVariable = value;
    }
```

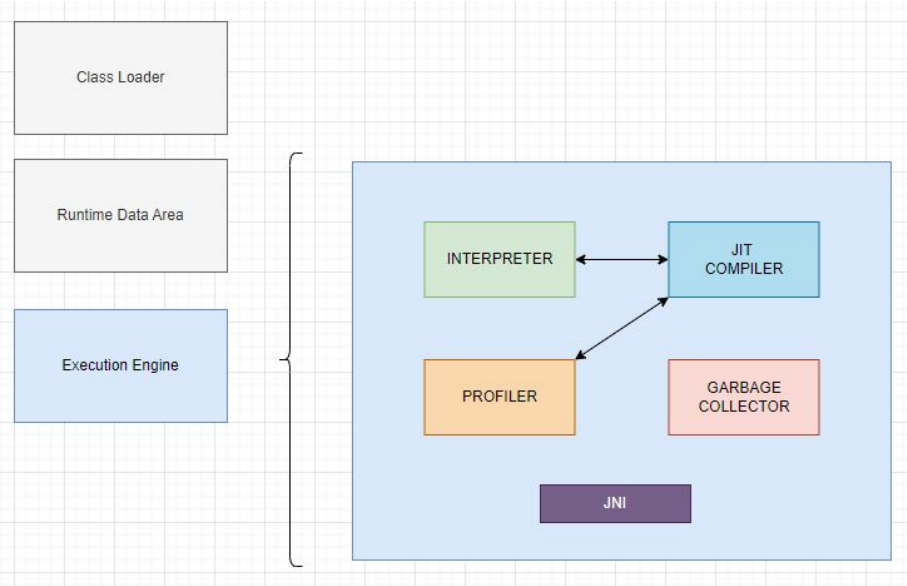
STACK MEMORY -> Local variables, operand stack, frame data (catch)

```java
public class StackMemory {

    public static void main(String[] args) {
        int a = 10; // Local variable 'a'
        int b = 5;  // Local variable 'b'

        int sum = add(a, b); // Method invocation with arguments (references)
        System.out.println("Sum: " + sum);
    }

    public static int add(int num1, int num2) {
        int result = 0; // Local variable 'result'

        try {
            result = num1 + num2; // Addition operation
        } catch (Exception e) { // Frame data
            System.out.println("Exception caught: " + e.getMessage());
        }

        return result;
    }
}
```

PC REGISTERS -> Current executions instructions

NATIVE METHOD STACK -> Native methods, JNI, Memory management

- Execution engine



INTERPRETER -> Read bytecode and executes step by step

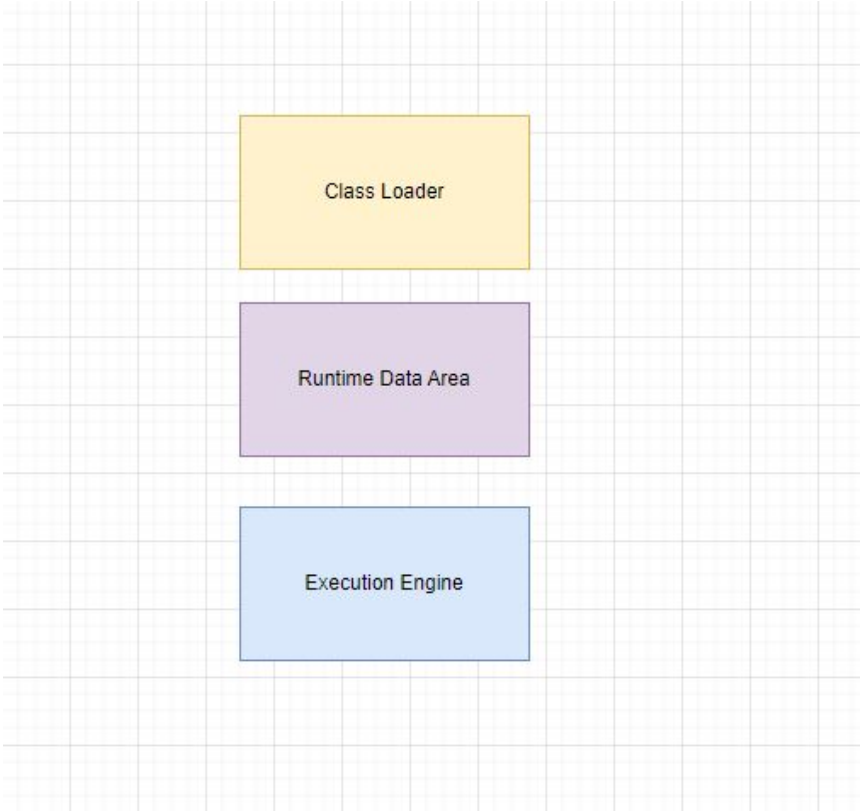
JIT COMPILER -> compiles bytecode to native code for repeated method calls
- Intermediate code generator
- Code optimizer
- Target code generator (intermediate -> native machine)

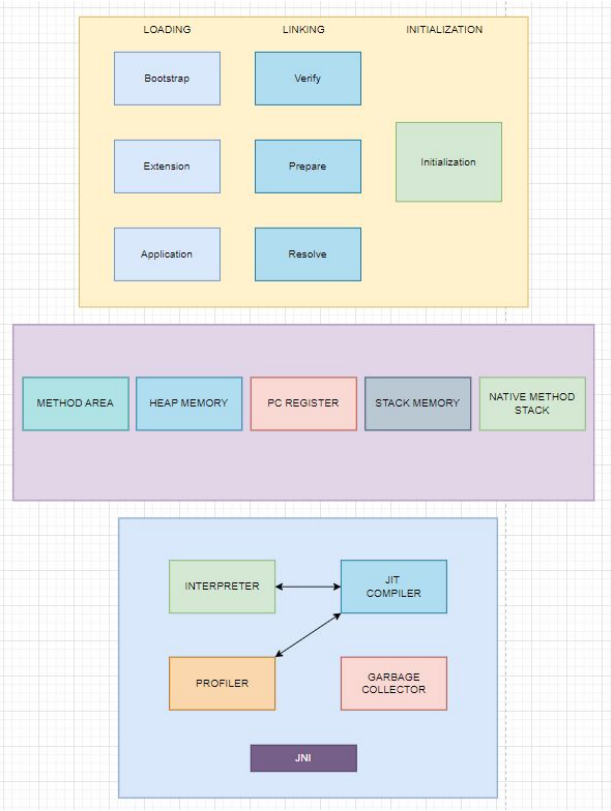
PROFILER -> find hotspots

GARBAGE COLLECTOR -> destroys the objects that are no longer used

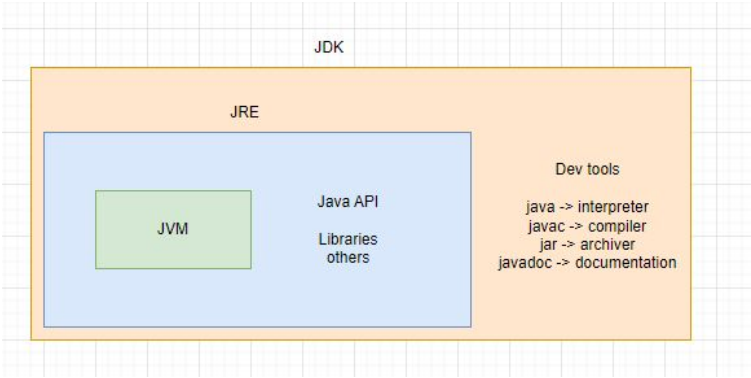
JNI -> Java Native Interface

JVM Architecture





JVM, JRE, JDK, J… AVA world

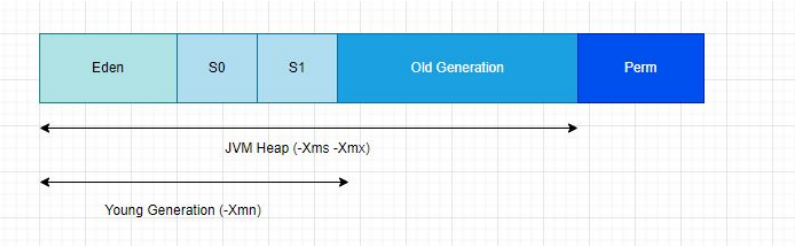


JVM -> Java Virtual Machine
- specification
- implementation
- runtime instance (java -> instantiating a new JVM)

JRE -> Java Runtime Environment
- includes JVM
- precompiled classes JCL
- JIT
- Platform independence

JDK -> Java Development Kit
- includes JRE
- includes dev tools:
  - java, javac
  - jar
  - JConsole, jstat
  - jmap
  - jdb
  - jmc
  - jstack

* JMM



Eden Memory -> new objects

Survivor memory -> moved by minor GC

Old generation -> survived some n generations

Perm -> app metadata

```
$ java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar myApp.jar
```

* GC

Minor GC -> promotes from Eden to S

Major GC -> promotes to old if obj survived multiple generations (by age threshold)

Stop the world -> timeout

Marking -> Deletion -> Compacting

Serial GC (-XX:+UseSerialGC)
Parallel GC (-XX:+UseParallelGC)
Parallel Old GC (-XX:+UseParallelOldGC)
Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)
G1 Garbage Collector (-XX:+UseG1GC)

Tips

Use Efficient Data Structures: (ArrayList instead of Vector if synchronization is not needed)

Limit Object Creation: Minimize unnecessary object creation, especially in performance-critical sections of your code.

Optimize Garbage Collection: fine-tune garbage collection settings

Manage Large Data Sets: consider using streaming or paging techniques to process data in smaller, manageable chunks

Avoid Memory Leaks: Make sure to release resources properly, close connections, and nullify object references when they are no longer needed

Profile and Monitor Memory Usage: Use profiling tools to analyze and identify memory hotspots or areas of excessive memory usage in your application

Use Caching: Utilize caching mechanisms to store frequently accessed data in memory, reducing the need for repeated calculations or expensive I/O operations

Properly Tune JVM Settings: such as heap size (-Xmx and -Xms) and garbage collection options (-XX:MaxGCPauseMillis, -XX:GCTimeRatio, etc.)

Regularly Monitor and Analyze Performance: Continuously monitor your application's memory usage and performance using monitoring tools and profiling techniques.

Thank you!

Lucian Gruia
lugr@ciklum.com

luciangruia.ro