JVM Performance and Tuning

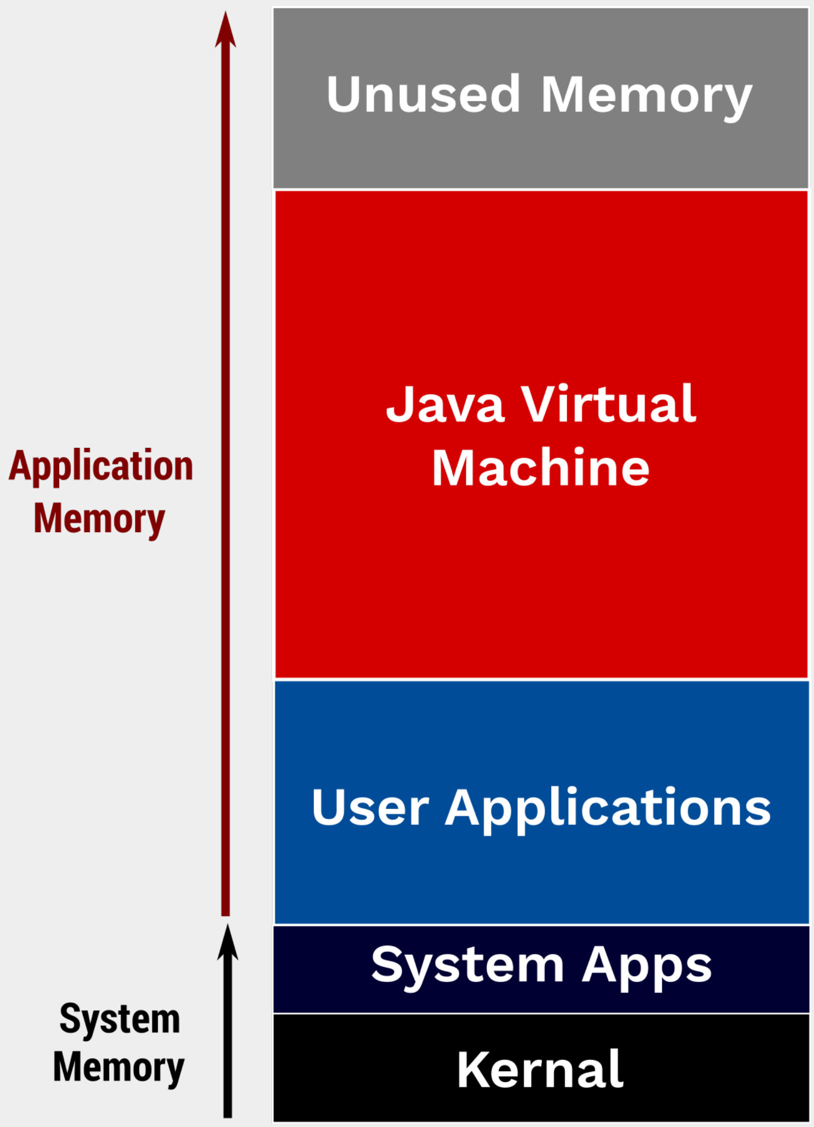
Module Three

JVM Architecture

Module Topics

1. The JVM Architecture
2. The Class Loader
3. JVM Memory
4. JVM Execution Engine
5. Modern Hardware

The Java Virtual Machine

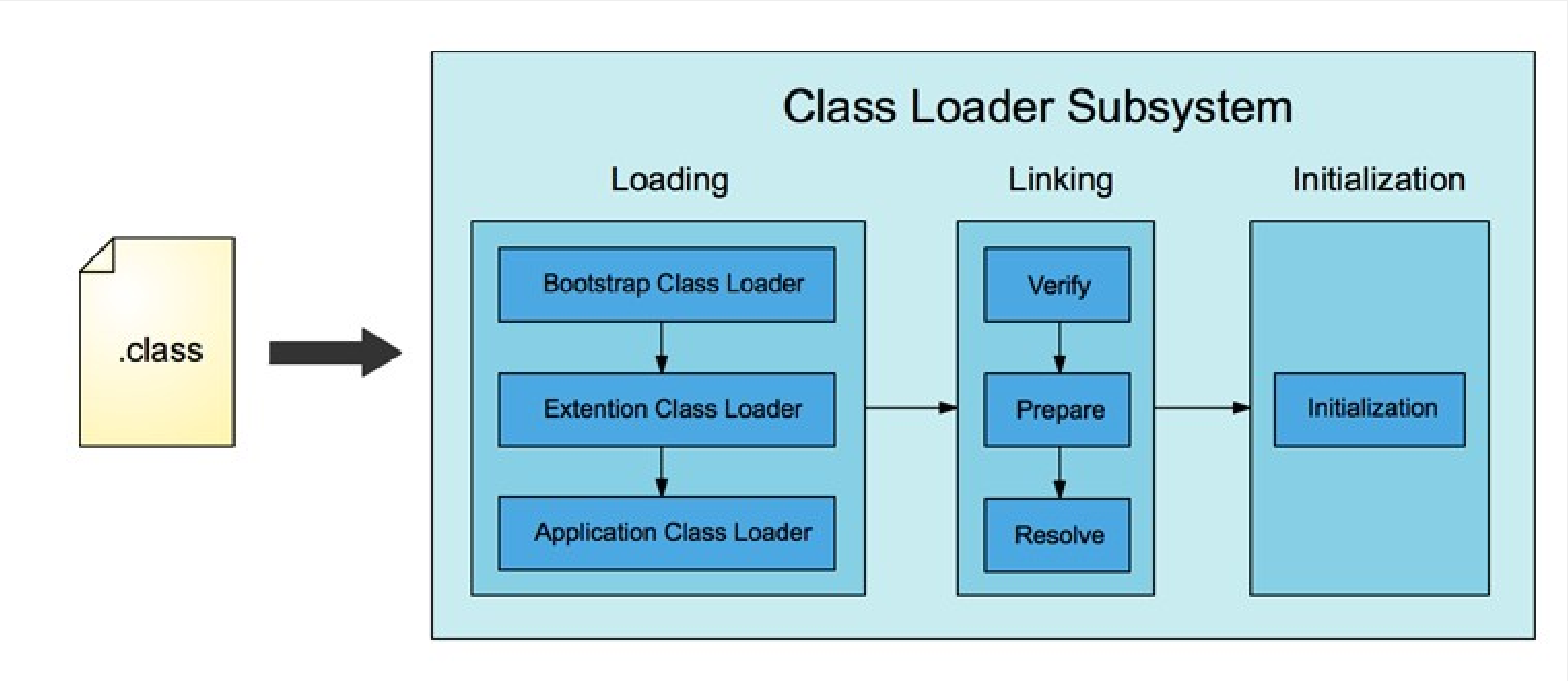


- The JVM is actually an application that runs as a program on a host Os
- JVM performance depends on:
  - Capability of host OS
  - How the host OS allocates resources among user applications

Module Topics

1. The JVM Architecture
2. The Class Loader
3. JVM Memory
4. JVM Execution Engine
5. Modern Hardware

The Class Loader



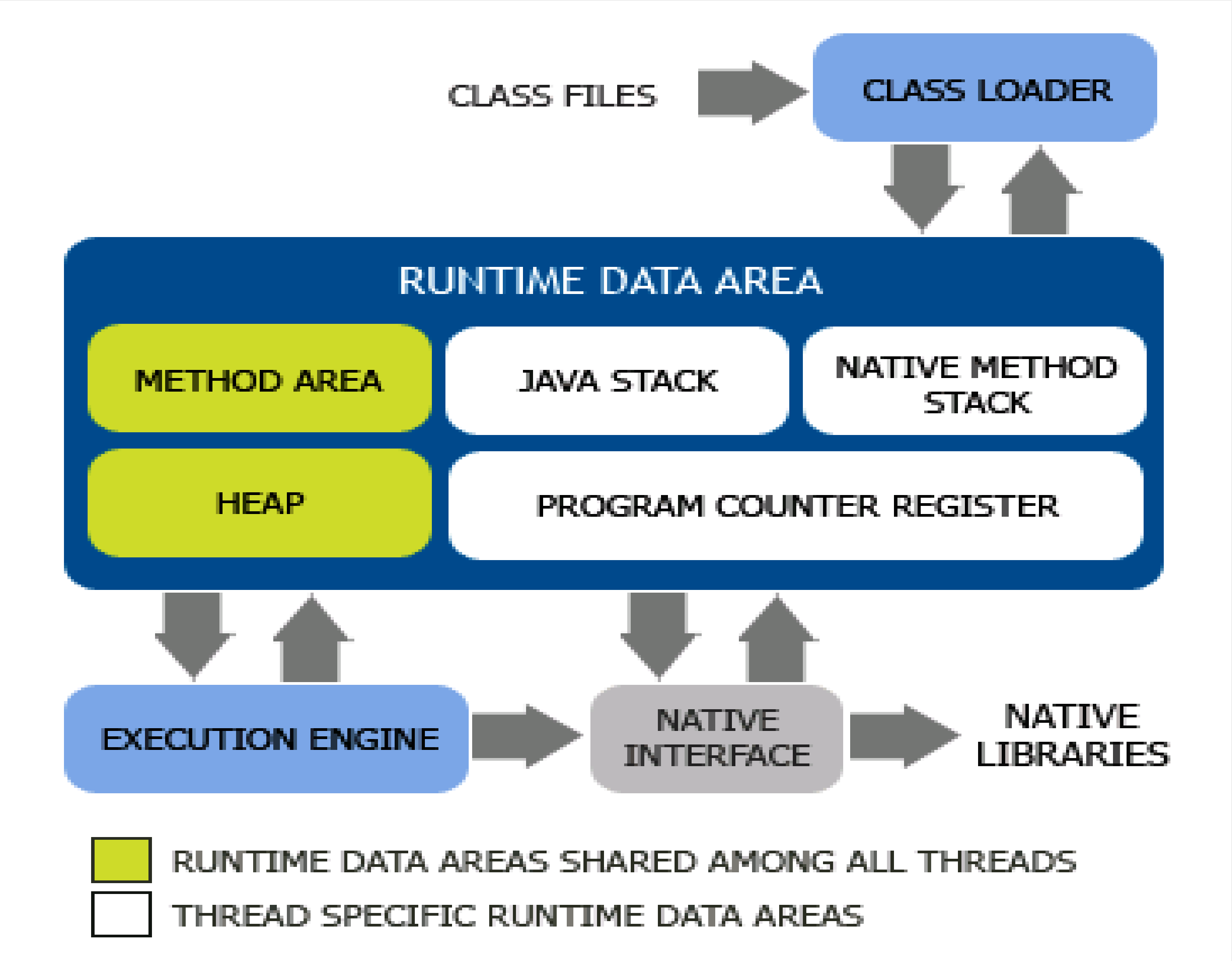
Class Loader and Performance

- Class loader is responsible for loading, linking and initialization
- Not a significant contributor to performance
- Except for start up time.

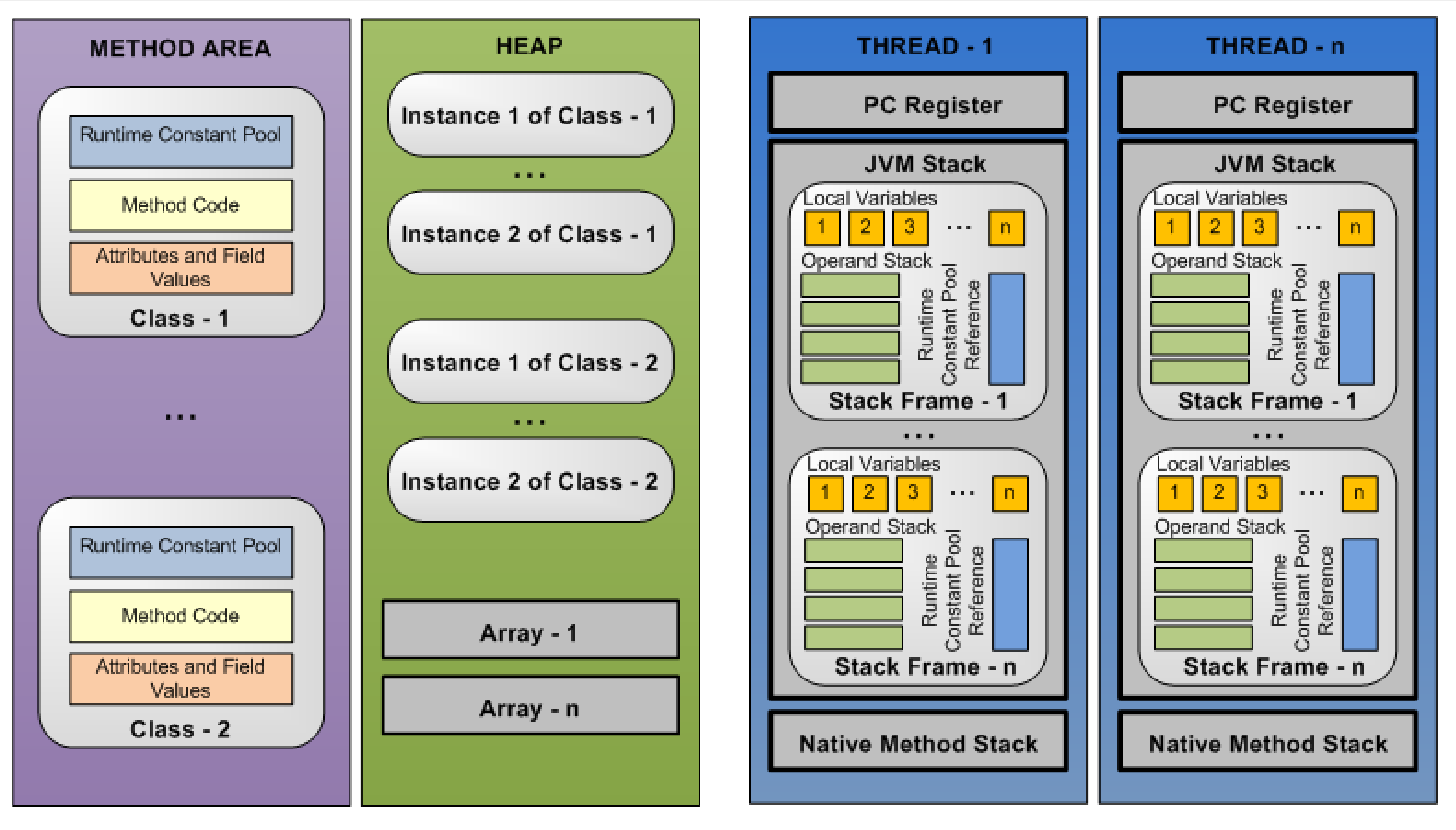
Module Topics

1. The JVM Architecture
2. The Class Loader
3. JVM Memory
4. JVM Execution Engine
5. Modern Hardware

The JVM Architecture

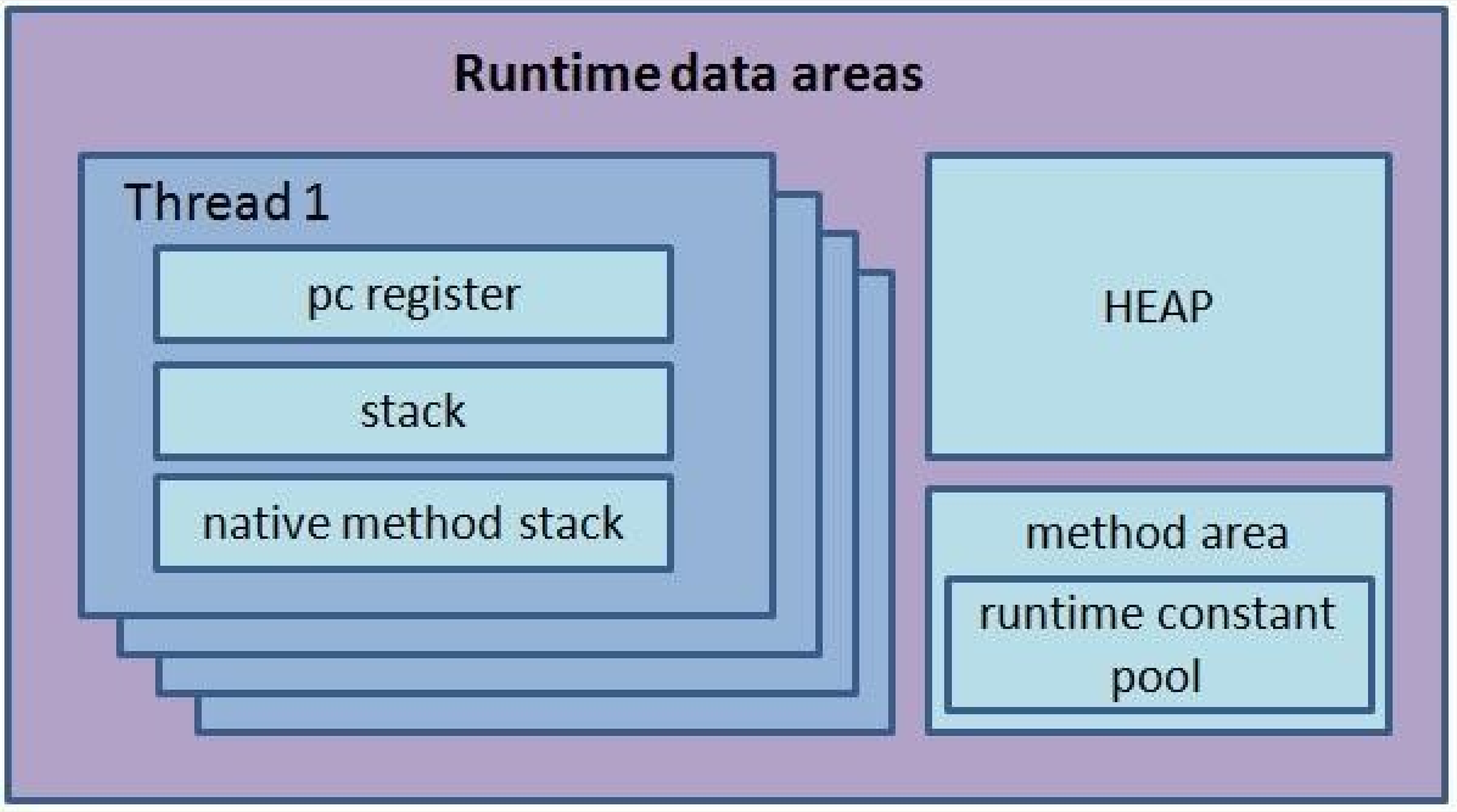


JVM Memory



METHOD AREA
Runtime Constant Pool
Method Code
Attributes and Field Values
Class - 1
...
Runtime Constant Pool
Method Code
Attributes and Field Values
Class - 2
HEAP
Instance 1 of Class - 1
...
Instance 2 of Class - 1
Instance 1 of Class - 2
...
Instance 2 of Class - 2
Array - 1
Array - n
THREAD - 1
PC Register
JVM Stack
Local Variables
1 2 3 ... n
Operand Stack
Runtime Constant Pool Reference
Stack Frame - 1
...
Local Variables
1 2 3 ... n
Operand Stack
Runtime Constant Pool Reference
Stack Frame - n
Native Method Stack
THREAD - n
PC Register
JVM Stack
Local Variables
1 2 3 ... n
Operand Stack
Runtime Constant Pool Reference
Stack Frame - 1
...
Local Variables
1 2 3 ... n
Operand Stack
Runtime Constant Pool Reference
Stack Frame - n
Native Method Stack

JVM Memory



Runtime data areas
Thread 1
pc register
stack
native method stack
HEAP
method area
runtime constant pool

Memory Areas

- Method Area
  - Shared data area that stores per-class structures
    - *The constant pool, field and method data, and the code for methods and constructors.*
    - *Memory management is automatic – under control of JVM*
- Heap
  - Shared data area that stores all data objects
    - *Including both class instances and arrays*
    - *Allocated when explicitly requested*
    - *Deallocated by JVM garbage collector automatically.*

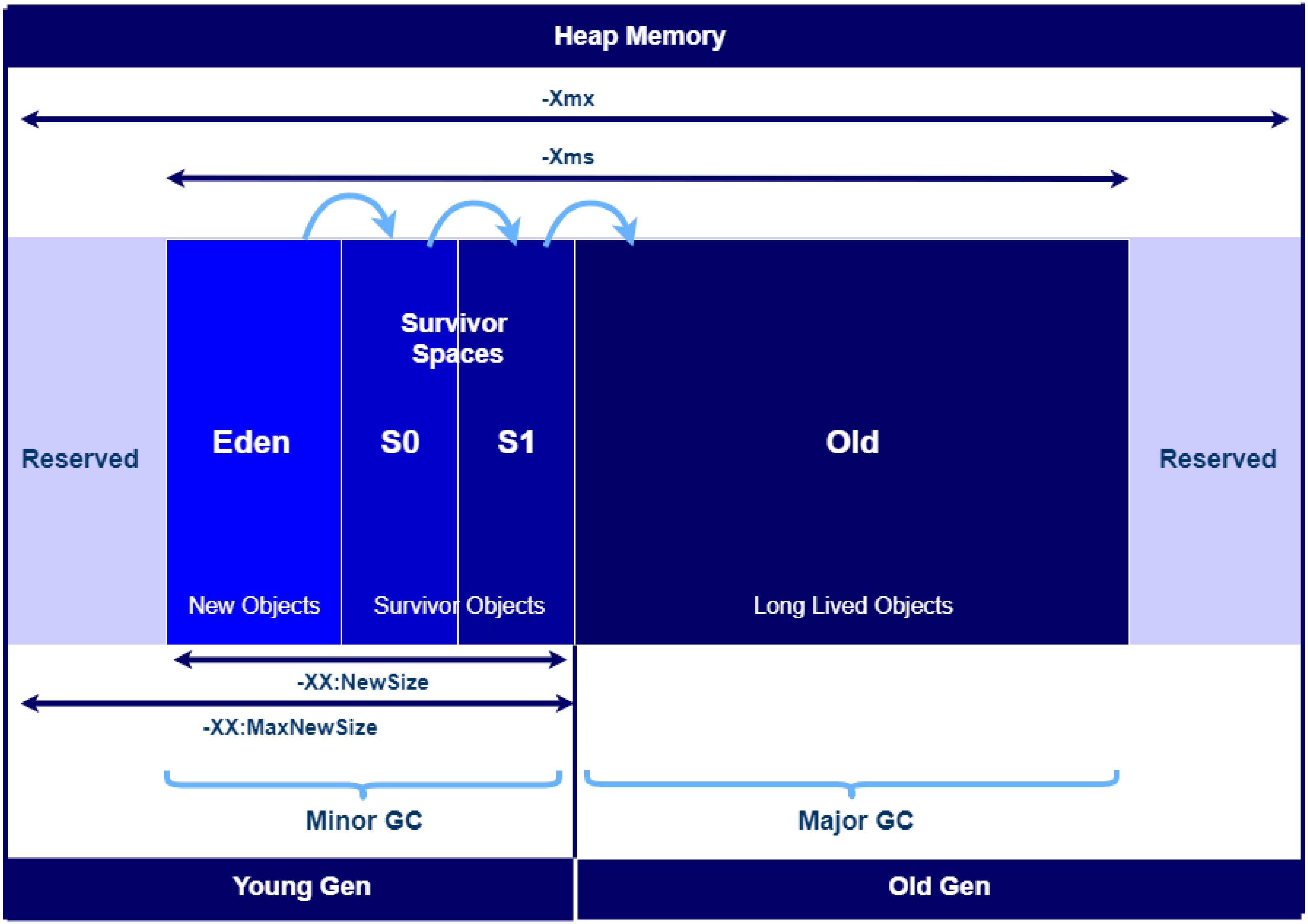
Memory Areas

- Direct Memory
  - Shared data area that stores buffer objects
    - *Allocated explicitly in direct memory area.*
    - *Automatically deallocated by JVM garbage collector.*
- PC (Program Counter) Register
  - Per-thread data area that contains the address of the instruction currently being executed in the thread.
  - If the method currently being executed is native, the value of the PC register is undefined.

Memory Areas

- JVM Stack
  - Per-thread data area that stores a stack of frames
  - A frame holds local variables and partial results
  - Used in method invocation and return of current method
- Native Method Stack
  - Per-thread data area that stores similar data elements
  - Like JVM stack but used to execute native (non-Java) methods.

Heap Memory Areas



JVM Heap Memory
(Image: PlatformEngineer.com)

Heap Memory Areas

- Heap is divided into 2 parts
  - Young Generation and Old Generation
- Heap is allocated when JVM starts up
  - (Initial size: -Xms)
- Heap size increases/decreases while the application is running
  - Maximum size: -Xmx

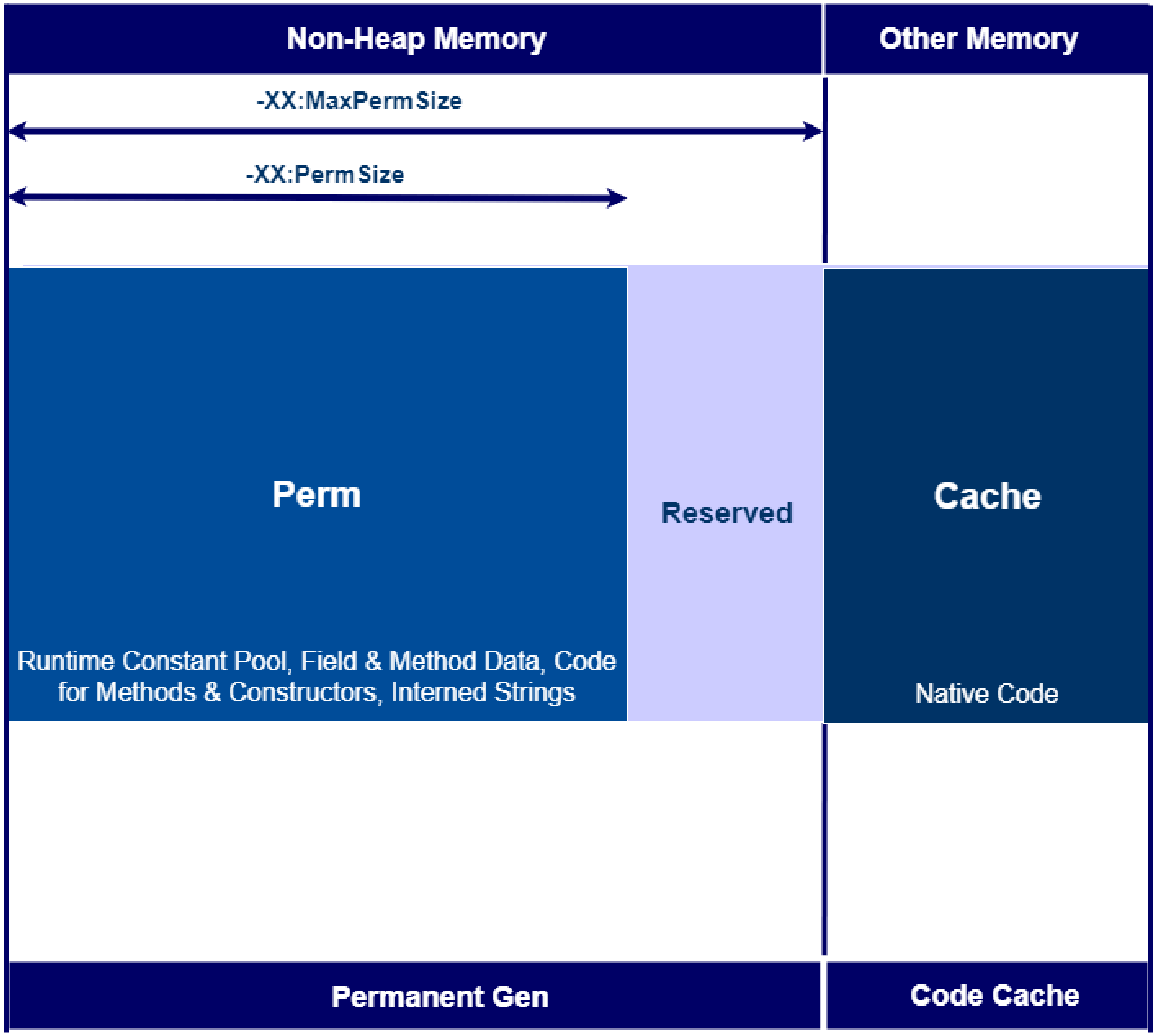
Young Generations

- Reserved for newly-allocated objects
  - Includes three parts
    - *Eden Memory*
    - *Two Survivor Memory spaces (S0, S1)*
  - Most of the newly-created objects goes Eden space.
  - When Eden space is filled Minor GC (a.k.a. Young Collection) is performed
    - *All the survivor objects are moved to one of the survivor spaces.*
    - *Minor GC also checks the survivor objects and move them to the other survivor space*
    - *One of the survivor space is always empty.*
  - Objects that are survived after many cycles of GC, are moved to the Old generation memory space

Old Generation

- Reserved for containing long lived objects that survive after many rounds of Minor GC
- When Old Gen space is full, Major GC (a.k.a. Old Collection) is performed
  - Usually takes a long time
- The purpose of generations is to make GC more efficient
  - Ties frequency of GC to likelihood an object needs to be removed

Non–Heap Memory Areas

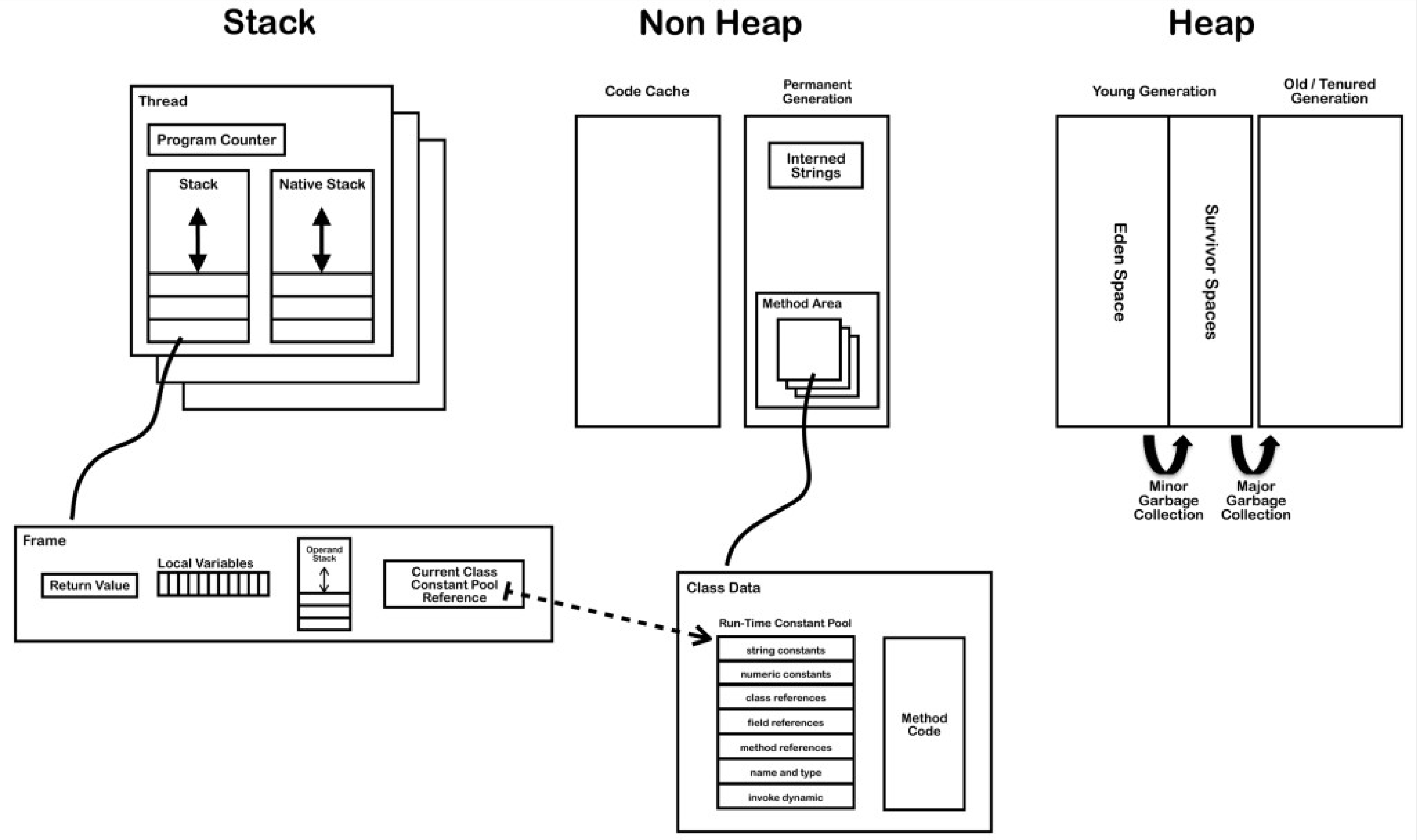


JVM Non-Heap & Cache Memory
(Image: PlatformEngineer.com)

Non–Heap Memory

- Perm Gen stores per-class structures such as:
  - runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings
  - Size can be changed using -XX:PermSize and -XX:MaxPermSize
- Cache Memory
  - Stores compiled code (i.e. native code) generated by JIT compiler, JVM internal structures, loaded profiler agent code and data, etc.
  - When Code Cache exceeds a threshold, it gets flushed (and objects are not relocated by the GC

Memory Integration



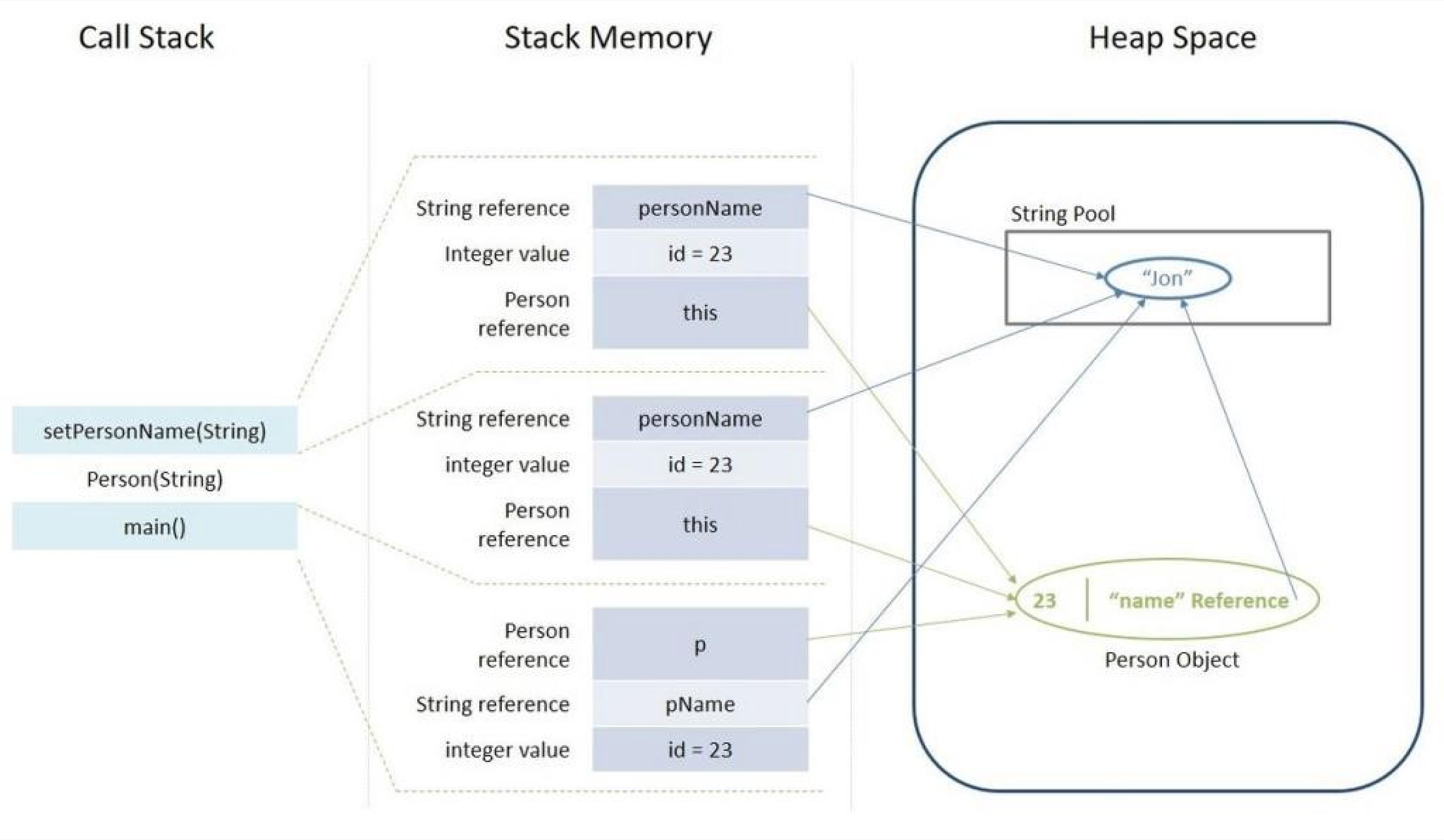
Memory Integration

```
class Person {
    int pid;
    String name;

// constructor, setters/getters
}

public class Driver {
    public static void main(String[] args) {
        int id = 23;
        String pName = "Jon";
        Person p = null;
        p = new Person(id, pName);
    }
}
```

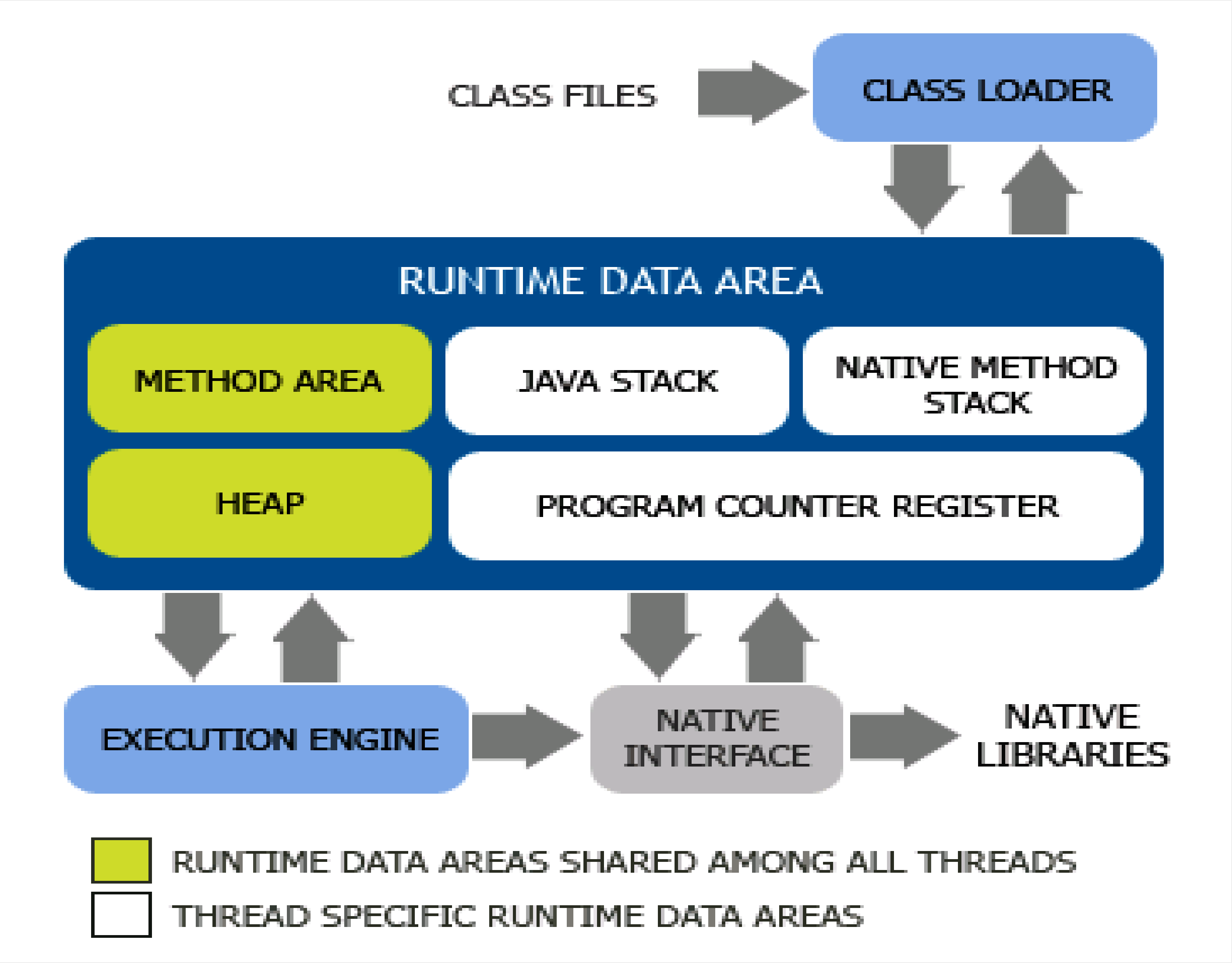
Memory Integration



Module Topics

1. The JVM Architecture
2. The Class Loader
3. JVM Memory
4. **JVM Execution Engine**
5. Modern Hardware

The JVM Architecture



Classic Java Interpreter

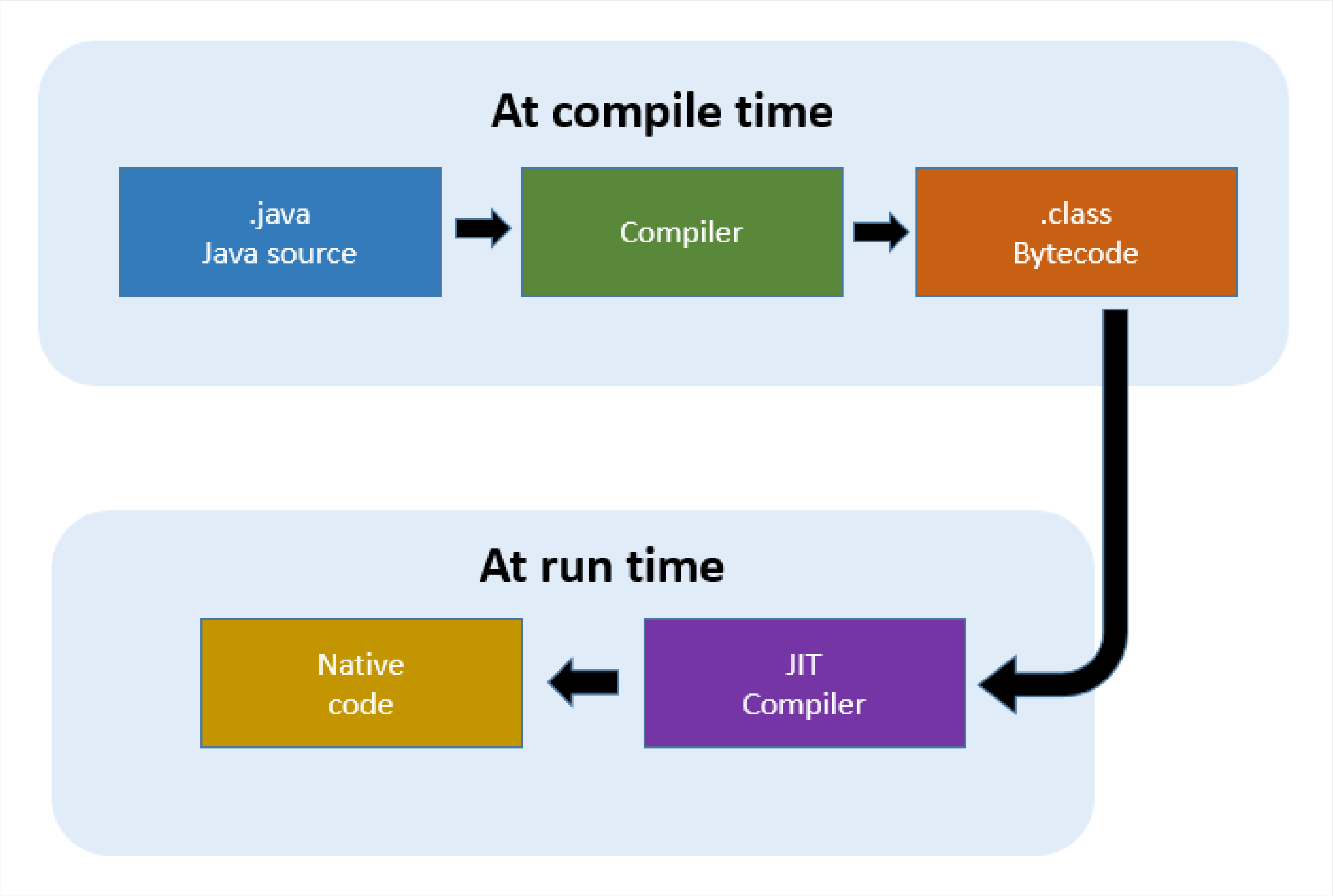
- Interpreted model
  - Each opcode in a class file is executed in turn
  - Classic interpreter
  - Not particularly efficient but runs anywhere there is an interpreter

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

Just in Time (JIT) Compiler

- Compiles each line of code
  - Optimizes the code
  - Caches the compiled code for later use
  - The more code that is compiled, the faster the execution become
- Profiles your code as it runs
  - The JIT compiler uses JVM resources
  - Used because expected improvements offset any resources used by the KIT compiler

JIT Compilation



Ahead of Time (AOT) Compiler

- Compiles all of the code before it is run
  - Like a standard compiler
- Use Cases for AOT
  - To speed up less used code that doesn’t get JIT attention
  - Environments where JIT is impractical
    - *Embedded systems*
- AOT output is platform specific – not portable

Module Topics

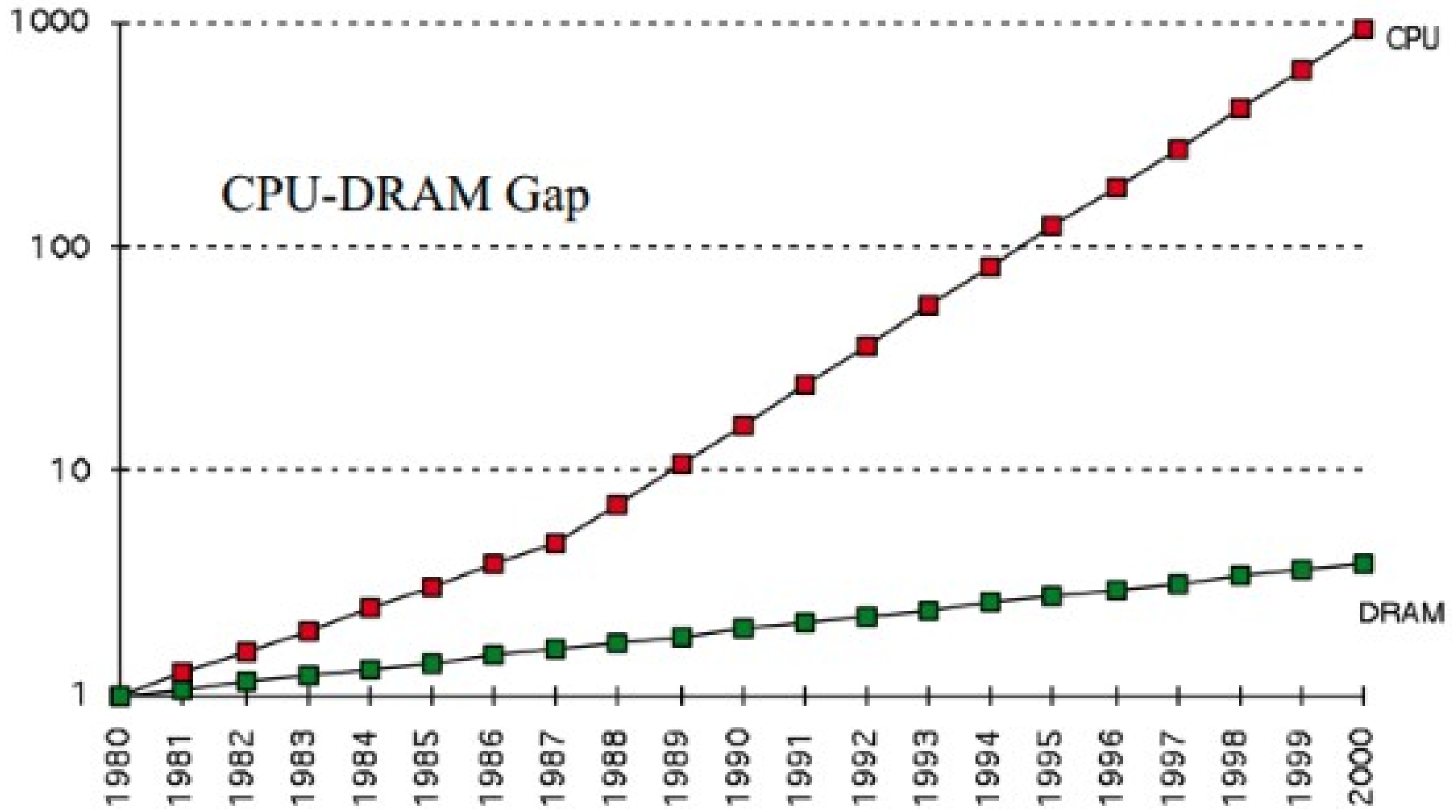
1. The JVM Architecture
2. The Class Loader
3. JVM Memory
4. JVM Execution Engine
5. Modern Hardware

Modern Architectures

- Java is a 1990s vintage architecture
- Life for a CPU was simpler
- Hardware improvements have been exponential
  - CPU architectures have been radically transformed
  - How CPUs execute code is radically different

CPU versus Memory

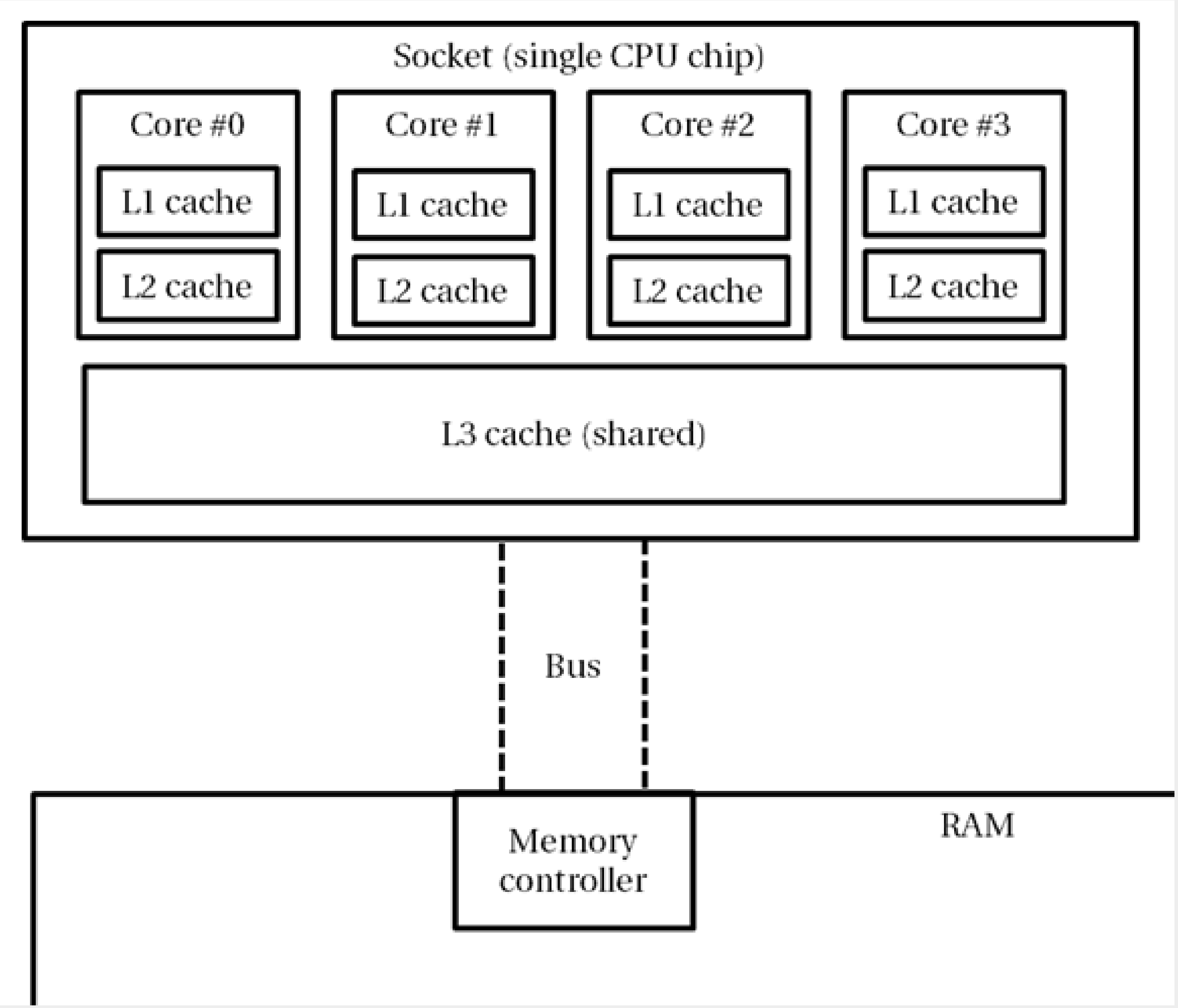
- Processor vs Memory Performance



1980: no cache in microprocessor;

1995 2-level cache

Modern CPU Architectures



Memory Caching

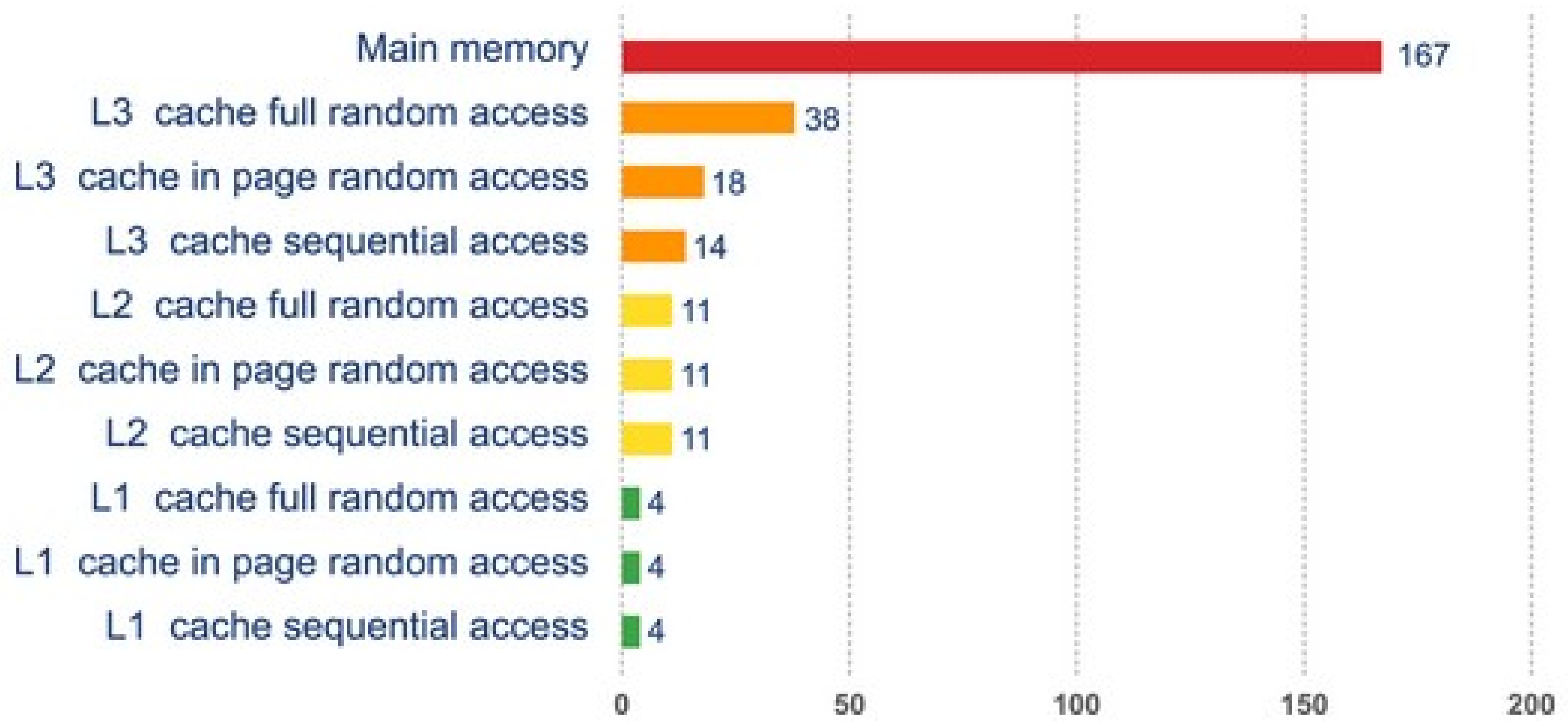


Figure 3-2. Access times for various types of memory

Modern Architectures

- CPU applies different strategies to improve performance
  - Translation Lookaside Buffer
  - Branch Prediction
  - Speculative Execution
- All of these work below the level of what we can tune in a JVM

Module End