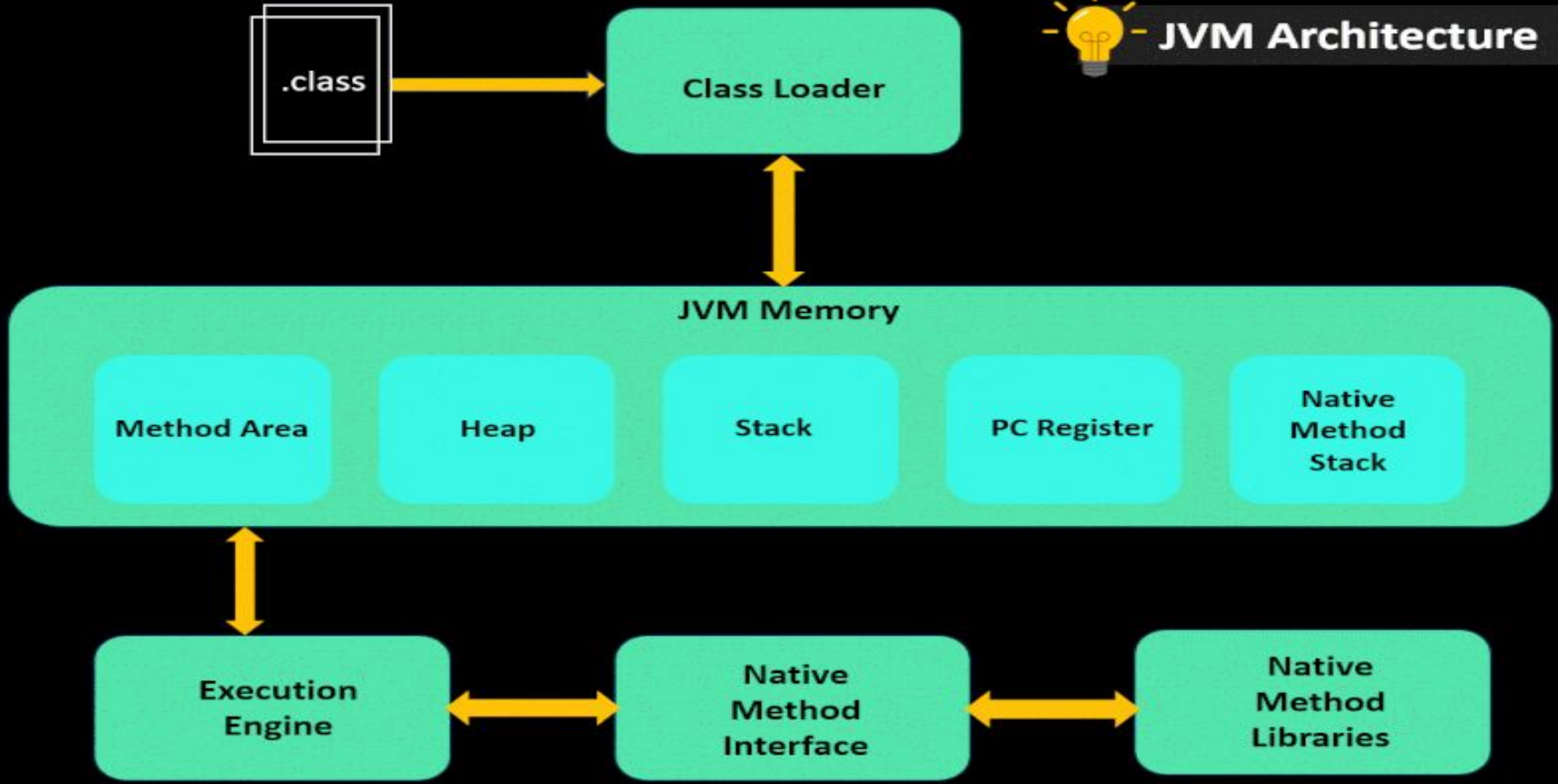
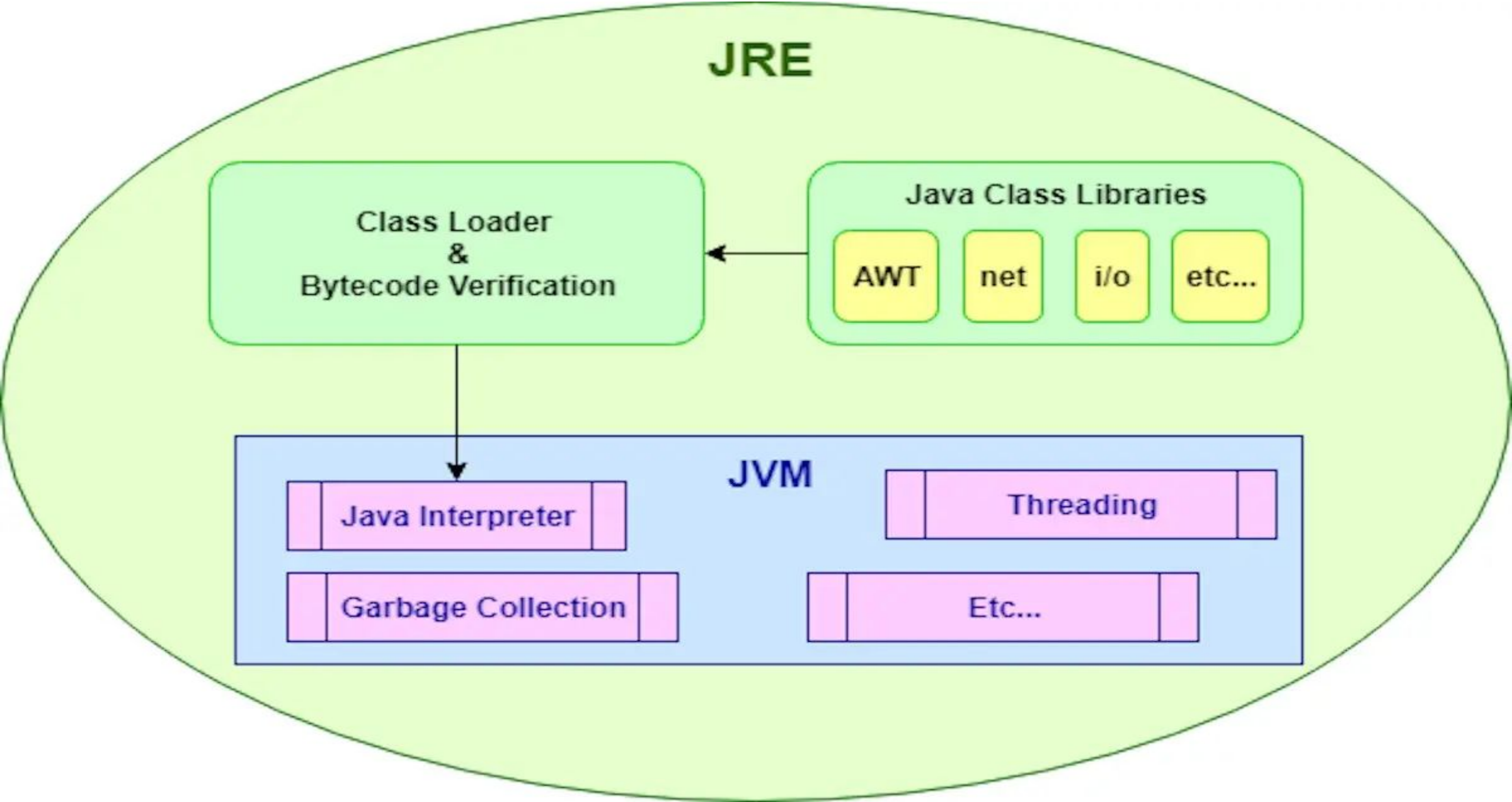
Java Virtual Machine

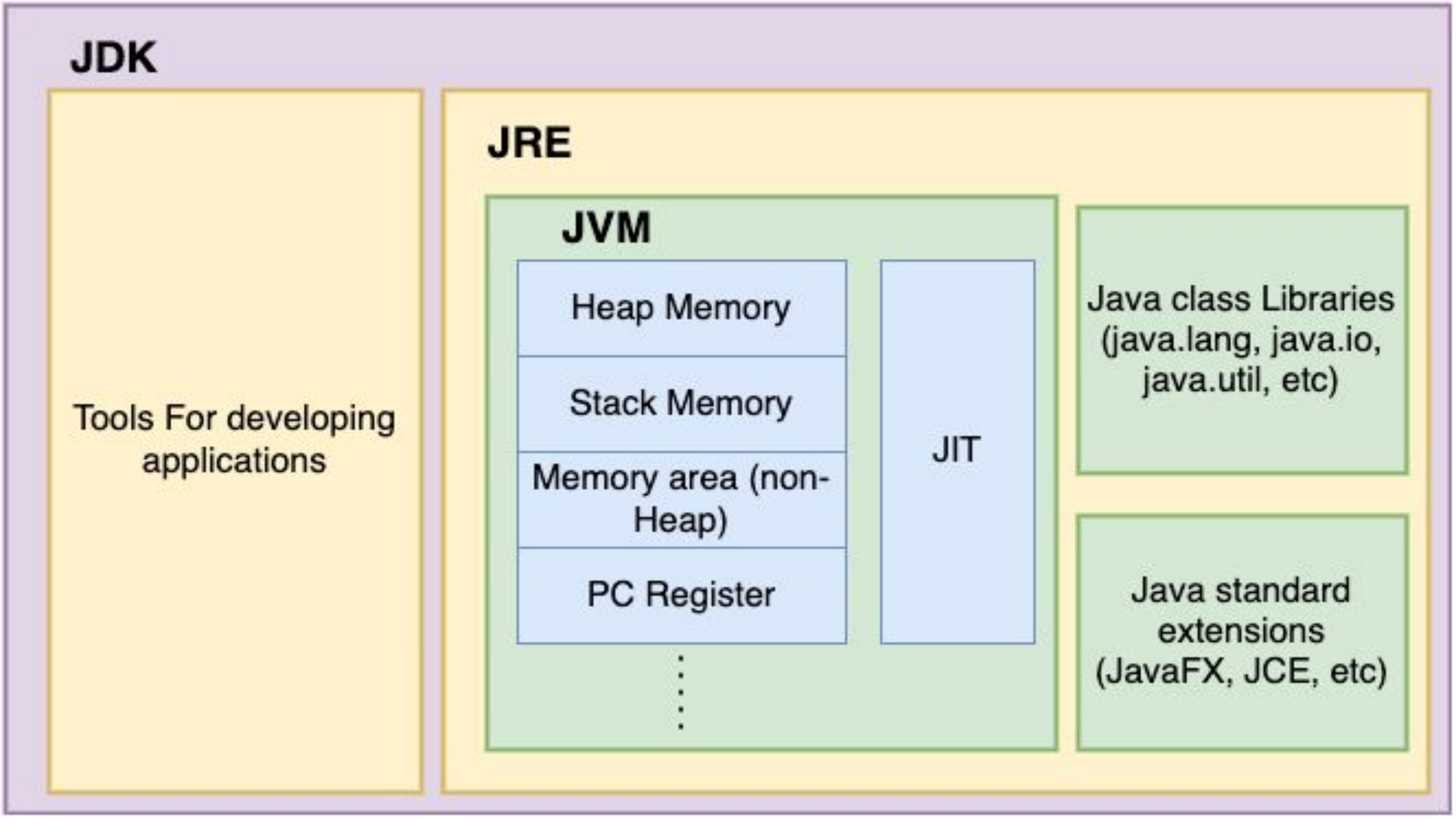

groovy
JRuby
Jython
Jython
Scala
Clojure
Kotlin

Language	Designed For	Key Features	Interoperability with Java
Java	General-purpose, enterprise	Strongly typed, OOP, vast ecosystem	Native
Kotlin	Android, modern Java alt.	Null safety, concise syntax, coroutines	Full (seamless)
Scala	Functional & OOP programming	Immutable data, pattern matching, type inference	Full
Groovy	Scripting, DSLs, build tools	Dynamic typing, closures, metaprogramming	Full
Clojure	Functional, Lisp dialect	Immutable data, macros, concurrency	Good (via Java interop functions)
JRuby	Ruby on JVM	Ruby syntax, metaprogramming, dynamic typing	Moderate
Jython	Python on JVM	Python syntax, Java library access	Moderate
Golo	Lightweight dynamic language	Simple syntax, prototype-based OOP	Limited
Xtend	Java alternative, DSLs	Type inference, lambda expressions, compiles to Java	Full

JVM Architecture




JRE
Class Loader
&
Bytecode Verification
Java Class Libraries
AWT
net
i/o
etc...
JVM
Java Interpreter
Threading
Garbage Collection
Etc...

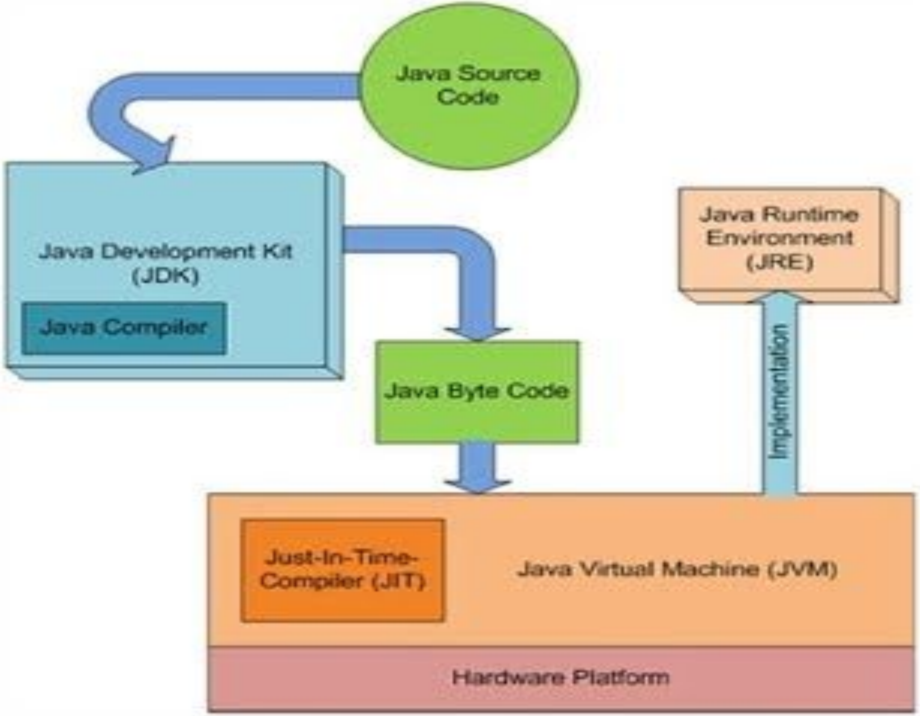

JDK
Tools For developing applications
JRE
JVM
Heap Memory
Stack Memory
Memory area (non-Heap)
PC Register
JIT
Java class Libraries (java.lang, java.io, java.util, etc)
Java standard extensions (JavaFX, JCE, etc)

Difference between JDK/JRE/JVM/JIT

JVM: Java Virtual Machine (JVM) is an abstract computing machine. Java programs are written against the JVM specification. JVM is specific for OS platform and they translate the Java instructions to the underlying platform specific instructions and execute them. JVM enables the Java programs to be platform independent.

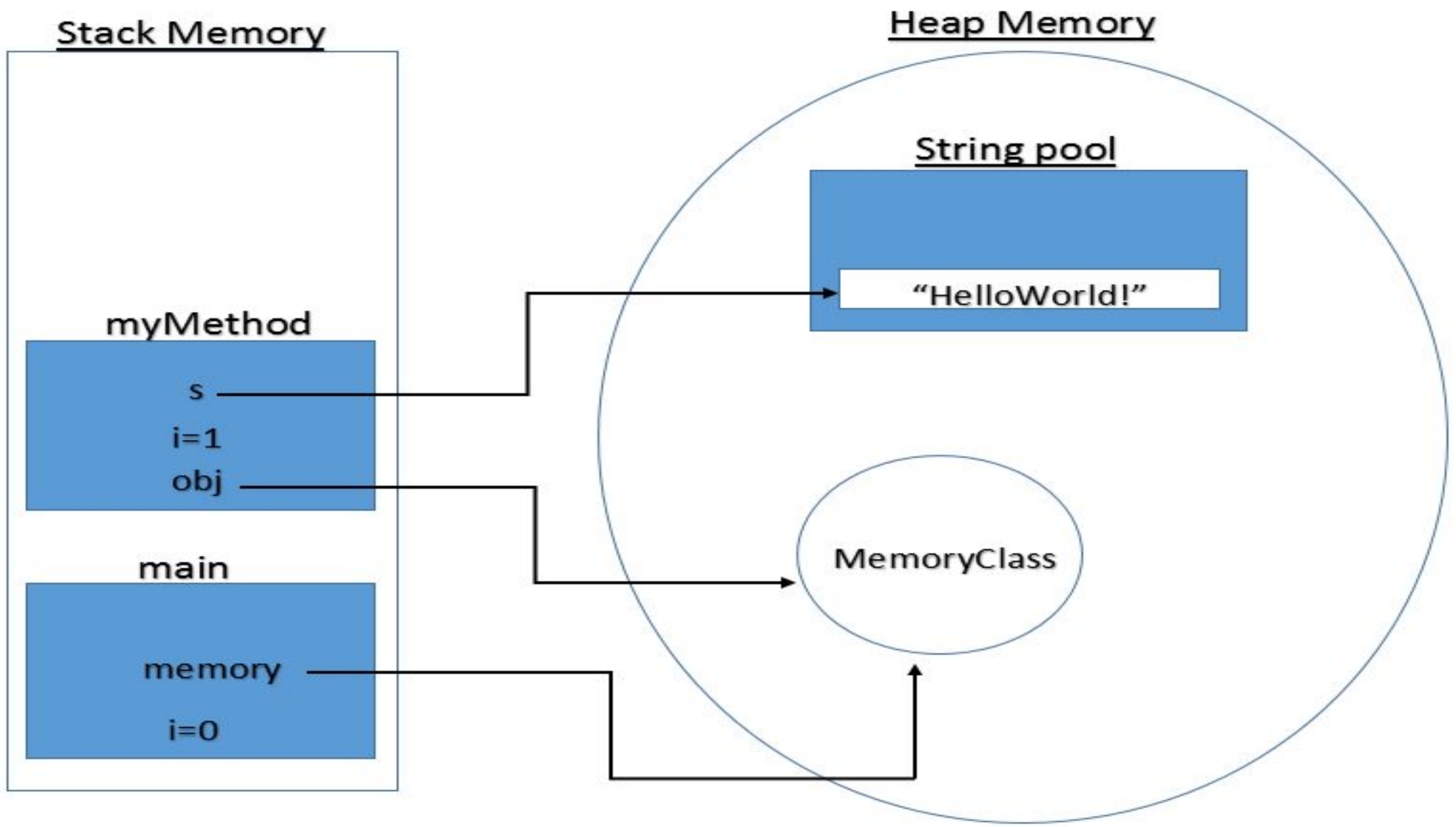
JRE: Java Runtime Environment (JRE) is an implementation of the JVM and Java API.

JDK: Java Development Kit (JDK) contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools.



The stack and heap are both areas of memory used in programming, but they have different purposes and characteristics.

	Stack	Heap
Purpose	Stores local variables and function parameters	Stores larger, longer-lived objects
Size	Fixed size, can overflow	No fixed size, can store unlimited data
Accessibility	Only accessible by the function that created it	Globally accessible
Lifetime	Limited to the function in which it was created	Can outlive the function in which it was created
Allocation	Fast and local to a function call	Slower and explicitly allocated by your program



Stack Memory
Heap Memory
String pool
"HelloWorld!"
myMethod
s
i=1
obj
main
memory
i=0
MemoryClass

Impact on Programming

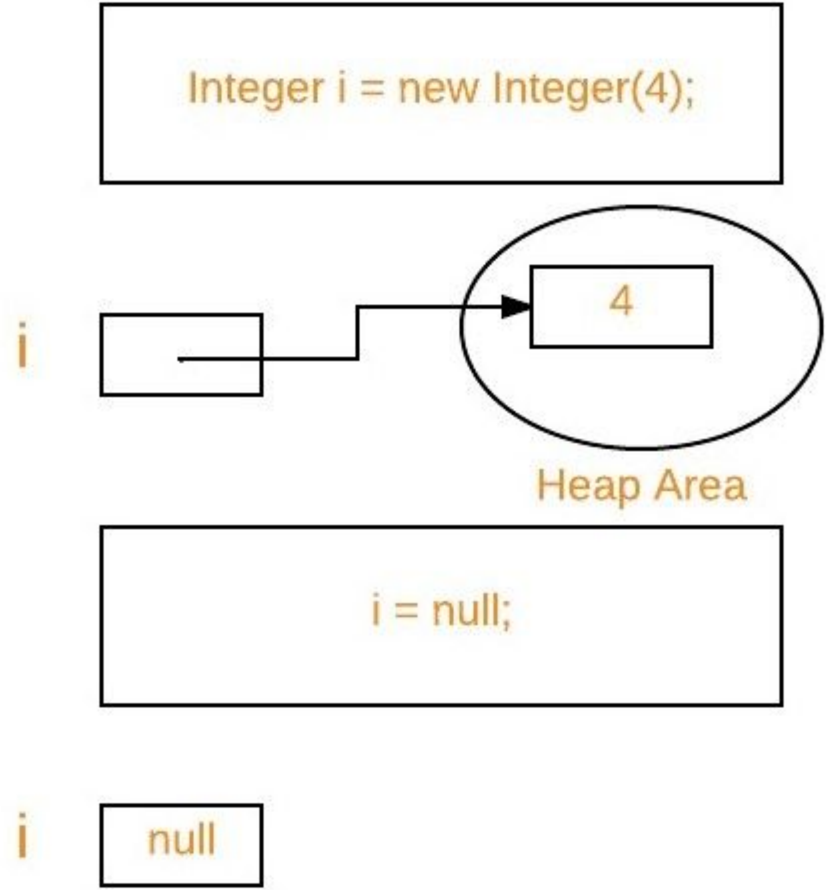
- **Garbage Collection (GC) overhead:** Objects in the heap are managed by GC, meaning developers don't manually free memory, but inefficient object management can cause **OutOfMemoryError** or performance issues.
- **Memory fragmentation:** Since objects are dynamically allocated, fragmentation can occur, leading to inefficient memory usage.
- **Longer access time:** Accessing heap memory is slower than stack memory due to pointer dereferencing and GC overhead.

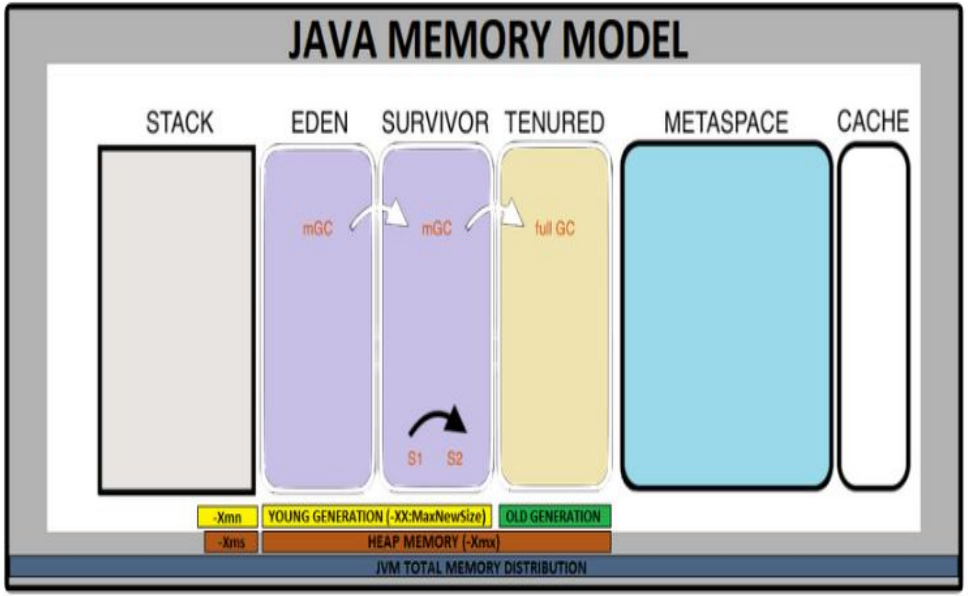
https://www.youtube.com/watch?v=3KqaI7QaCCM

Optimizing JVM Performance via Stack & Heap

- **Minimize object creation:** Reduce heap pressure by reusing objects (e.g., using `StringBuilder` instead of concatenation).
- **Use primitive types when possible:** This reduces heap allocations since primitives reside in the stack.
- **Beware of memory leaks:** Improper object references (like static collections holding unnecessary objects) can prevent GC from reclaiming memory.
- **Tune JVM heap settings:** Use options like `-Xms` (initial heap size) and `-Xmx` (maximum heap size) to optimize performance.

Java Garbage Collection





- In the image shown above for Java Memory Model:
  - Heap Space: Eden + Survivor + Tenured
  - Non-Heap Space: Stack + MetaSpace + Reserved (Not shown here)
  - Cache

Memory Space	Description	Used For	GC Behavior
Eden Space (E)	Part of the Young Generation	Newly allocated objects	Frequent Minor GC (objects that survive move to Survivor)
Survivor 0 (S0)	Part of the Young Generation	Holds objects surviving one GC cycle	Objects are moved between S0 and S1 or promoted to Old Gen
Survivor 1 (S1)	Part of the Young Generation	Alternate survivor space	Objects are copied between S0 ↔ S1 during GC
Old Generation (O)	Long-lived objects	Objects promoted from Young Gen	Full GC occurs when Old Gen fills up
Metaspace (M)	Stores class metadata, method data, JIT-compiled code	Classloading, method data, JIT optimizations	Dynamically resizes, no GC (but can be cleaned up)
Compressed Class Space (CCS)	Holds class metadata in a compressed form	Used to optimize Metaspace memory usage	Cleared when classes are unloaded
Code Cache	Stores JIT-compiled machine code	Optimized execution of frequently used code	Managed by JVM, not subject to standard GC

```
  501  6746  6214   0  9:10PM ttys001    0:00.00 grep java
jonwhite@Jons-MBP ~ % jps
6769 Jps
6733 GCDemo
jonwhite@Jons-MBP ~ % jstat -gc 6733 1000
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     CGC    CGCT     GCT
 0.0   1024.0  0.0   1024.0 1694720.0   0.0    2253824.0  2047525.3  4864.0 3487.0 512.0  308.9   1616    2.238   0      0.000 3232     1.983    4.221
 0.0   1024.0  0.0   1024.0 1684480.0   0.0    2264064.0  2047525.3  4864.0 3487.0 512.0  308.9   1634    2.262   0      0.000 3268     2.005    4.267
 0.0   1024.0  0.0   1024.0 1712128.0   0.0    2236416.0  2047525.3  4864.0 3487.0 512.0  308.9   1656    2.287   0      0.000 3312     2.030    4.317
 0.0   1024.0  0.0   1024.0 1701888.0   0.0    2246656.0  2047525.3  4864.0 3487.0 512.0  308.9   1674    2.307   0      0.000 3348     2.051    4.358
 0.0   1024.0  0.0   1024.0 1714176.0   0.0    2234368.0  2047525.3  4864.0 3487.0 512.0  308.9   1695    2.330   0      0.000 3390     2.077    4.407
 0.0   1024.0  0.0   1024.0 1718272.0   0.0    2230272.0  2047525.3  4864.0 3487.0 512.0  308.9   1713    2.351   0      0.000 3426     2.097    4.449
 0.0    0.0    0.0    0.0   2488320.0 1024.0  1461248.0   765472.5  4864.0 3487.0 512.0  308.9   1732    2.374   0      0.000 3464     2.119    4.493
 0.0   1024.0  0.0   1024.0 1694720.0   0.0    2253824.0  2047525.3  4864.0 3487.0 512.0  308.9   1758    2.407   0      0.000 3516     2.149    4.556
 0.0   1024.0  0.0   1024.0 1692672.0   0.0    2255872.0  2047525.3  4864.0 3487.0 512.0  308.9   1775    2.426   0      0.000 3550     2.170    4.595
 0.0   1024.0  0.0   1024.0 1864704.0   0.0    2083840.0  1822245.3  4864.0 3487.0 512.0  308.9   1795    2.449   0      0.000 3590     2.197    4.646
 0.0   1024.0  0.0   1024.0 1710080.0   0.0    2238464.0  2047525.3  4864.0 3487.0 512.0  308.9   1814    2.472   0      0.000 3628     2.226    4.697
 0.0   1024.0  0.0   1024.0 1598464.0   0.0    2350080.0  2047525.3  4864.0 3487.0 512.0  308.9   1825    2.519   0      0.000 3650     2.308    4.827
 0.0   1024.0  0.0   1024.0 1712128.0   0.0    2236416.0  2047525.3  4864.0 3487.0 512.0  308.9   1831    2.535   0      0.000 3662     2.318    4.853
 0.0   1024.0  0.0   1024.0 1708032.0   0.0    2240512.0  2047525.3  4864.0 3487.0 512.0  308.9   1854    2.565   0      0.000 3708     2.344    4.909
 0.0   1024.0  0.0   1024.0 1708032.0   0.0    2240512.0  2047525.3  4864.0 3487.0 512.0  308.9   1873    2.591   0      0.000 3745     2.366    4.957
 0.0    0.0    0.0    0.0   2296832.0 1024.0  1658880.0  1662496.5  4864.0 3487.0 512.0  308.9   1891    2.611   0      0.000 3782     2.391    5.001
 0.0   1024.0  0.0   1024.0 1701888.0   0.0    2246656.0  2047525.3  4864.0 3487.0 512.0  308.9   1911    2.633   0      0.000 3822     2.413    5.046
 0.0   1024.0  0.0   1024.0 1701888.0   0.0    2246656.0  2010661.3  4864.0 3487.0 512.0  308.9   1934    2.658   0      0.000 3868     2.446    5.104
 0.0   1024.0  0.0   1024.0 1716224.0   0.0    2232320.0  2047525.3  4864.0 3487.0 512.0  308.9   1953    2.681   0      0.000 3906     2.472    5.153
 0.0    0.0    0.0    0.0   2488320.0 1024.0  1461248.0   417312.5  4864.0 3487.0 512.0  308.9   1972    2.708   0      0.000 3944     2.497    5.205
 0.0    0.0    0.0    0.0   2488320.0 1024.0  1461248.0  1037856.5  4864.0 3487.0 512.0  308.9   1991    2.732   0      0.000 3982     2.519    5.252
 0.0   1024.0  0.0   1024.0 1727488.0   0.0    2221056.0  1998373.3  4864.0 3487.0 512.0  308.9   2013    2.761   0      0.000 4026     2.547    5.309
 0.0   1024.0  0.0   1024.0 1692672.0   0.0    2255872.0  2047525.3  4864.0 3487.0 512.0  308.9   2034    2.783   0      0.000 4068     2.571    5.354
 0.0   1024.0  0.0   1024.0 1701888.0   0.0    2246656.0  2047525.3  4864.0 3487.0 512.0  308.9   2055    2.809   0      0.000 4110     2.597    5.406
 0.0   1024.0  0.0   1024.0 1690624.0   0.0    2257920.0  2047525.3  4864.0 3487.0 512.0  308.9   2073    2.829   0      0.000 4146     2.618    5.446
 0.0   1024.0  0.0   1024.0 1864704.0   0.0    2083840.0  1936933.3  4864.0 3487.0 512.0  308.9   2091    2.858   0      0.000 4181     2.651    5.509
 0.0   1024.0  0.0   1024.0 1759232.0   0.0    2189312.0  2047525.3  4864.0 3487.0 512.0  308.9   2110    2.882   0      0.000 4220     2.672    5.554
 0.0   1024.0  0.0   1024.0 1690624.0   0.0    2257920.0  2047525.3  4864.0 3487.0 512.0  308.9   2130    2.904   0      0.000 4258     2.692    5.597
 0.0   1024.0  0.0   1024.0 1701888.0   0.0    2246656.0  2047525.3  4864.0 3487.0 512.0  308.9   2153    2.935   0      0.000 4306     2.719    5.654
 0.0   1024.0  0.0   1024.0 1710080.0   0.0    2238464.0  2047525.3  4864.0 3487.0 512.0  308.9   2171    2.955   0      0.000 4342     2.740    5.695
 0.0   1024.0  0.0   1024.0 1721344.0   0.0    2227200.0  1951269.3  4864.0 3487.0 512.0  308.9   2192    2.980   0      0.000 4384     2.765    5.745
 0.0   1024.0  0.0   1024.0 1694720.0   0.0    2253824.0  2047525.3  4864.0 3487.0 512.0  308.9   2210    3.001   0      0.000 4420     2.786    5.787
 0.0   1024.0  0.0   1024.0 1694720.0   0.0    2253824.0  2047525.3  4864.0 3487.0 512.0  308.9   2231    3.025   0      0.000 4462     2.810    5.835
 0.0   1024.0  0.0   1024.0 1692672.0   0.0    2255872.0  2047525.3  4864.0 3487.0 512.0  308.9   2249    3.045   0      0.000 4498     2.830    5.876
 0.0    0.0    0.0    0.0   2488320.0 1024.0  1461248.0   986656.5  4864.0 3487.0 512.0  308.9   2268    3.069   0      0.000 4536     2.853    5.922
```

Search Labs | AI Overview

The Mandelbrot set is computationally intensive because it requires iterating a complex mathematical function for each pixel in an image, and due to its fractal nature, the number of iterations needed to determine if a point belongs to the set can vary drastically depending on its location, often requiring a large number of calculations to accurately render intricate details, especially near the boundary of the set.

Key points about the computational intensity of the Mandelbrot set:

Fractal complexity:

The Mandelbrot set exhibits self-similarity, meaning that zooming into any part reveals similar patterns repeating at smaller scales, leading to an infinite level of detail that needs to be calculated for accurate rendering.

Iterative process:

To determine if a point belongs to the Mandelbrot set, a complex number is repeatedly squared and added to itself (iteration) until either it diverges to infinity (not in the set) or remains bounded (in the set).

Large number of iterations:

Depending on the location of a point, it can take a large number of iterations to determine if it belongs to the set, especially near the boundary where complex patterns emerge.

Pixel-by-pixel calculation:

To generate a visual representation of the Mandelbrot set, each pixel in the image needs to be individually calculated based on its corresponding complex number.