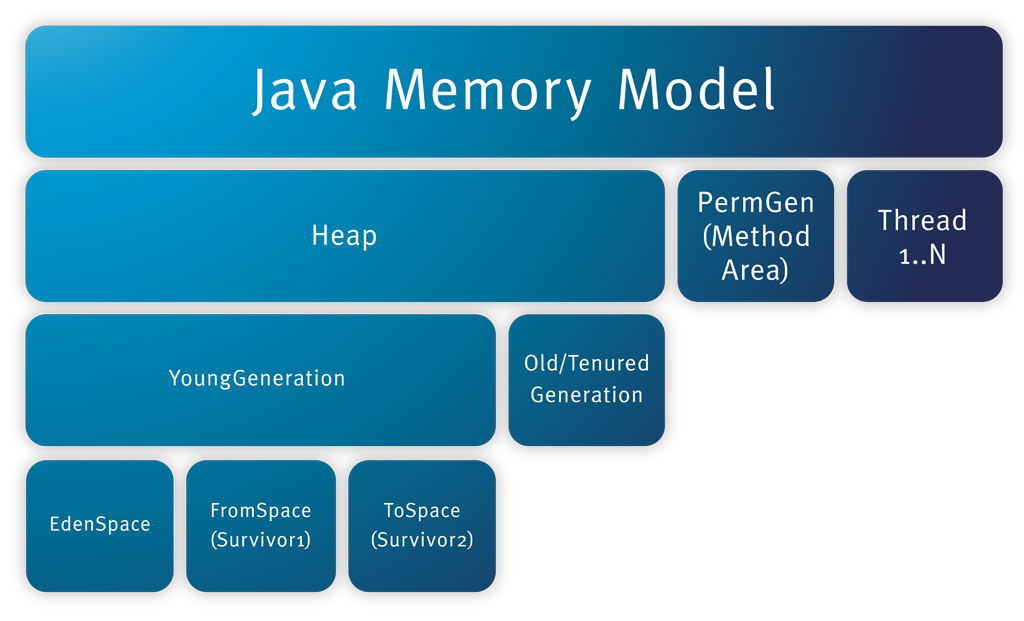
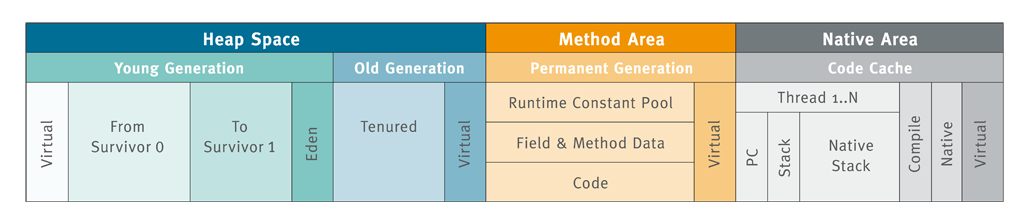
**JVM Memory System (HotSpot)**

1. Memory Model

The heap space holds object data, the method area holds class code, and the native area holds references to the code and object data.



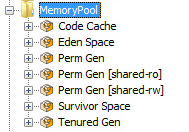
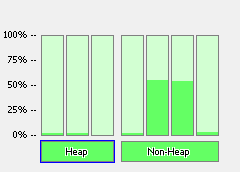


**Memory Pools and Memory Managers**

Memory pools and memory managers are key aspects of the Java VM's memory system.

* A **memory pool** represents a memory area that the Java VM manages. The Java VM has at least one memory pool and it may create or remove memory pools during execution. A memory pool can belong either to heap or to non-heap memory.
* A **memory manager** manages one or more memory pools. The garbage collector is a type of memory manager responsible for reclaiming memory used by unreachable objects. A Java VM may have one or more memory managers. It may add or remove memory managers during execution. A memory pool can be managed by more than one memory manager.

**JConsole Memory Pool Snippet:**



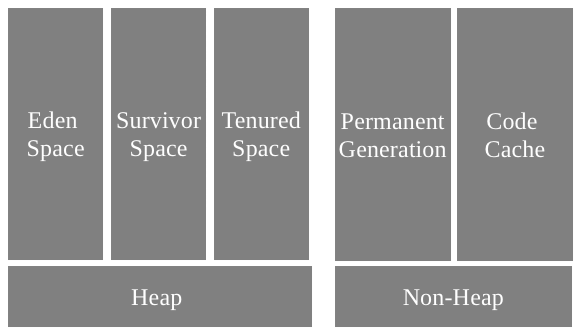
**Heap**: Eden Space, Survivor Space, and Tenured Gen

**Non-Heap**: Code Cache, Perm Gen [shared-ro], Perm Gen [shared-rw], and Perm Gen.

**Heap and Non-Heap Memory**

The Java VM manages two kinds of memory: heap and non-heap memory, both of which are created when the Java VM starts.

* **Heap memory** is the runtime data area from which the Java VM allocates memory for all class instances and arrays. The heap may be of a fixed or variable size. The garbage collector is an automatic memory management system that reclaims heap memory for objects.
  + 1. **Young / Nursery Generation**
       - Eden Space
       - Survivor Space
    2. **Tenured / Old Generation**
* **Non-heap memory** includes a method area shared among all threads and memory required for the internal processing or optimization for the Java VM. It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The method area is logically part of the heap but, depending on the implementation, a Java VM may not garbage collect or compact it. Like the heap memory, the method area may be of a fixed or variable size. The memory for the method area does not need to be contiguous.
  + 1. **Code Cache (Native Area)**
    2. **Permanent Generation (Method Area)**
       - Perm Gen [shared-read only]
       - Perm Gen [shared-read write]



**Eden Space**

The pool from which memory is allocated for most of the Objects.

Both the partial (minor) and full GC are done on Eden space.

**Survivor Space**

The pool contains the objects that have survived the garbage collection of the Eden space and the objects that require huge memory than Eden space.

Both the partial (minor) and full GC are done on Survivor space.

HotSpot VM maintains two Survivor spaces; one of the spaces (alternatively) will always be kept empty and used as swap area during garbage collection.

**Tenured Generation**

The pool contains objects that have existed for some time in the Survivor space.

Only full GC is done on Tenured Generation.

**Code Cache**

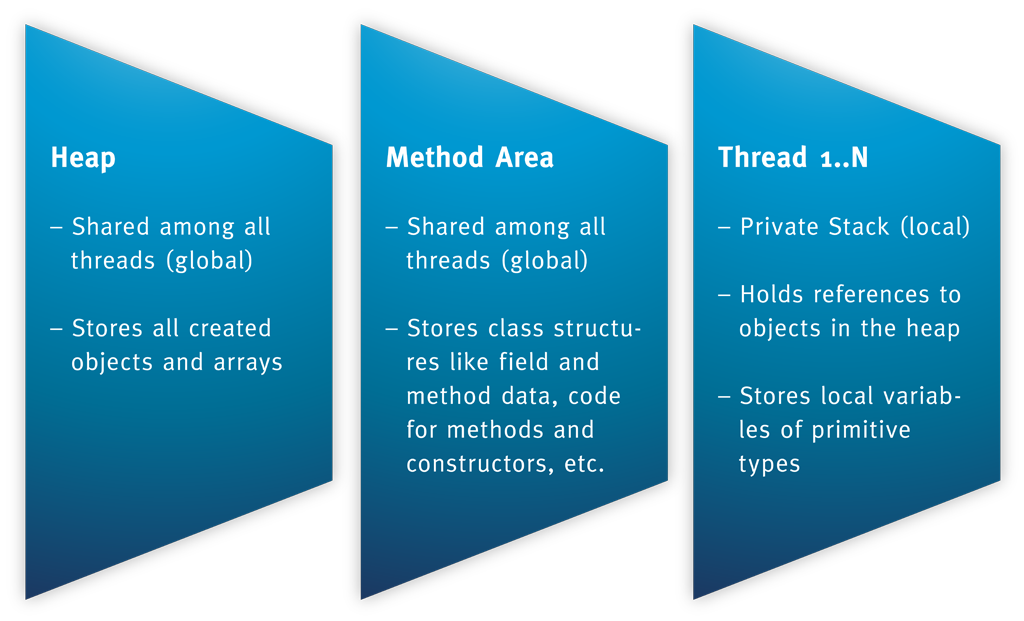
The pool used for compilation and storage of native code. It does involve in GC, but not the usual memory managers.

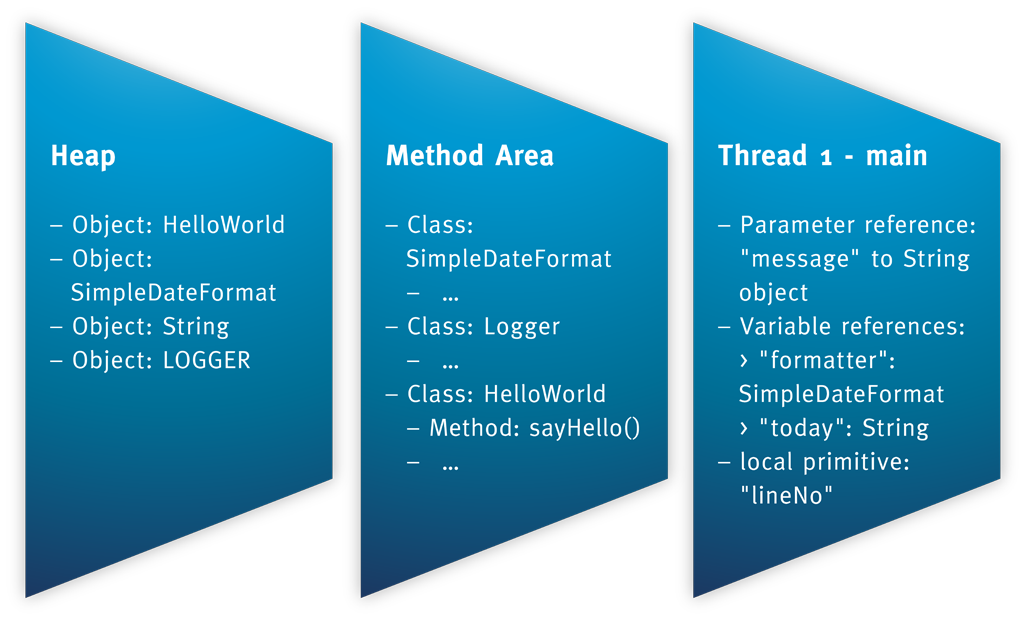
**Permanent Generation**

The pool contains all the reflective data of java virtual machine itself, such as runtime constant pool, method code, attribute, field, etc. In addition, interned Strings and class static variables are stored in the permanent generation. It is also called as method area.

With JVMs that use class data sharing, this generation is divided into two: read-only and read-write.

Only full GC is done on Permanent Generation.





**Memory Model Hierarchy (Heap Vs Non-Heap):**

**Memory Model Hierarchy (Private Vs Shared):**

**Runtime Data Area**

The Java virtual machine defines various runtime data areas that are used during execution of a program. They are shared and per thread data areas.

**Shared Data Area**

The data area shared among all the java virtual machine threads is called a shared data area.

This is created on Java virtual machine start-up and destroyed only when the Java virtual machine exits.

1. Heap
2. Method Area (Non Heap)

**Per Thread Data Area**

The data area private to each thread is called a per thread data area.

This is created when a thread is created and destroyed when the thread exits.

1. The pc register
2. JVM stacks
3. Native method stacks

**Heap**

Heap is the runtime data area (shared among all the Java VM threads) from which the Java VM allocates memory for all class instances and arrays.

The garbage collector reclaims heap memory for objects; objects are never explicitly deallocated.

The heap may be of a fixed or variable size, this is an implementation choice.

**Method Area (Perm Gen)**

Method area is the runtime data area, shared among all the Java VM threads.

It stores per-class structures such as the runtime constant pool, field, method data, and the code for methods and constructors, including the special methods used in class and instance initialization and interface initialization. In addition**, interned Strings and class static variables are stored in the permanent generation.**

Although the method area is logically part of the heap, simple implementations may choose not to either garbage collect or compact it.

**Runtime Constant Pool**

A runtime constant pool is a per-class or per-interface runtime representation of the constant\_pool table in a class file.

It contains several kinds of constants, ranging from numeric literals known at compile-time to method and field references that must be resolved at runtime.

Each runtime time constant pool is allocated from the JVM’s method area.

The runtime constant pool for a class or interface is constructed when a class or interface is created by the JVM.

**The pc Register**

Each JVM thread has its own pc (program counter) register. The pc register contains the address of the JVM instruction currently being executed, unless the method is native.

**Java Virtual Machine Stacks**

Each JVM thread has a private Java virtual machine stack, created at the same time as the thread. A Java virtual machine stack stores frames (a frame for a method).

**Native Method Stacks**

Java virtual machine may use conventional stacks, colloquially called "C stacks," to support native methods, methods written in a language other than the Java programming language.

**Frames**

A frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception).

Frames are allocated from the Java virtual machine stack of the thread creating the frame. Each frame has its own array of local variables, its own operand stack, and a reference to the runtime constant pool of the class of the current method.

1. Garbage Collection

Garbage collection (GC) is the process of releasing memory used by the dead objects; how the Java VM frees memory occupied by objects that are no longer referenced.

The objects that have active references are called "**alive**" and non-referenced (or unreachable) objects are called "**dead**".

The Java HotSpot VM garbage collector uses generational GC. Generational GC takes advantage of the observation that most programs conform to the following generalizations.

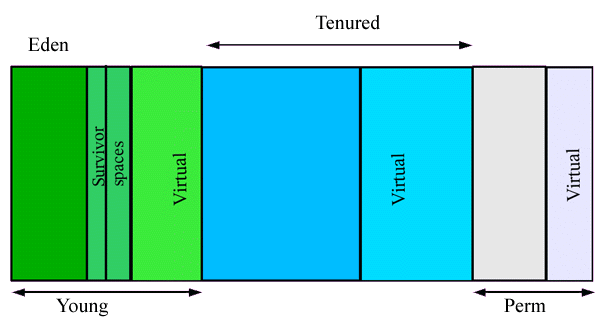
* They create many objects that have short lives, for example, iterators and local variables.
* They create some objects that have very long lives, for example, high level persistent objects.

The Java HotSpot VM defines two generations: the young generation (sometimes called the "nursery") and the old generation.

The **young generation** consists of an "Eden space" and two "survivor spaces." The VM initially assigns all objects to the Eden space, and most objects die there. When it performs a minor GC, the VM moves any remaining objects from the Eden space to one of the survivor spaces. The VM moves objects that live long enough in the survivor spaces to the "tenured" space in the **old generation**. When the tenured generation fills up, there is a full GC that is often much slower because it involves all live objects. The **permanent generation** holds all the reflective data of the virtual machine itself, such as class and method objects.

When a generation uses up its allotted memory, the VM performs a partial GC (also called a minor collection) on that memory pool to reclaim memory used by dead objects. This partial GC is usually much faster than a full GC.

**HotSpot VM Generations:**



1. String Literal Pool / String Internalization

String literal pool is a collection of references (weak references) to the String objects created at permanent generation memory.

A string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions - are "interned" so as to share unique instances, using the method String.intern.

String literal pool is a collection of objects.

A pool of strings, initially empty, is maintained privately by the class String or JVM.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the String.equals() method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

The **internalized strings are garbage collected** if there are no more references to them.

A few facts about string objects comparison result:

* The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.
* Literal strings within the same class in the same package represent references to the same String object.
* Literal strings within different classes in the same package represent references to the same String object.
* Literal strings within different classes in different packages likewise represent references to the same String object.
* Strings computed by constant expressions are computed at compile time and then treated as if they were literals.
* Strings computed by concatenation at run-time are newly created and therefore distinct.
* The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

**Where does String Pool get allocated in memory?**

The exact location of the string pool is not specified and can vary from one JVM implementation to another. It is interesting to note that until Java 7, the pool was in perm gen space on hotspot JVM but it has been moved to the heap since Java 7.

**In JDK6 or Earlier**: Interned strings are allocated in the permanent generation of the Java heap.

**In JDK 7**: Interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application.

**CR/Bug Fix Description at JDK7**: Interned strings are currently stored in the permanent generation. A new approach for managing meta-data is being designed and it requires interned strings to live elsewhere in the heap (young gen and/or old gen).

**Is String Literal Pool a collection of references to the String Object, or a collection of Objects itself?**

**Ans:** String literal pool is a collection of references (weak references) to the String objects created at permanent generation memory.

**“==” Vs equals**

* *Comparing strings with == is much faster than with String.equals()*, but in general it isn't near a performance improvement as it is cracked up to be.
* *String.intern() saves heap space*, but at the expense of using up the more precious Perm Gen space.
* *Internalized strings are released from memory* if they are no longer referenced.
* *String.intern() is dangerous* if you don't know what you are doing.

**Runtime Constant Poll Vs String Literal Pool**

A *runtime constant pool* is a per-class or per-interface runtime representation of the constant\_pool table in a class file. It contains several kinds of constants, ranging from numeric literals known at compile-time to method and field references that must be resolved at runtime.

A *string literal pool* is a collection of references (weak references) to the String objects created at permanent generation memory.

The literal Strings, and Strings resulted by constant expressions are computed at compile time itself and added to the *string literal pool* at runtime during class loading.

The internalized Strings are added to the *string literal pool* at runtime during execution.

1. Others

**Other JVMs (Proprietary/Free)**

* JRocket (Proprietary of Oracle)
* J9 (Proprietary of IBM)