**Heap:**

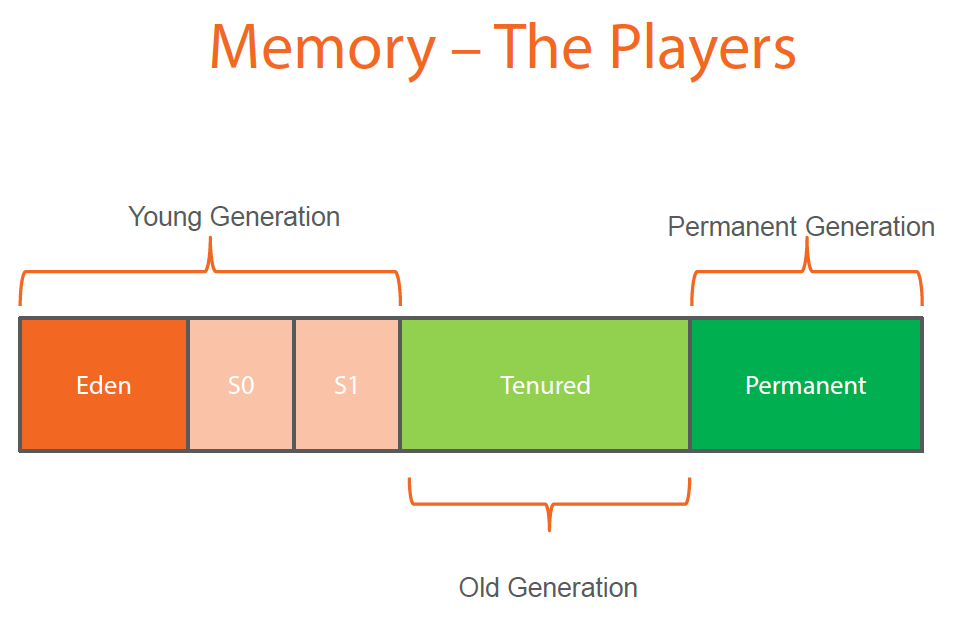
Primitive objects and their references are stored.

Static variables, objects and their references are stored.

**Young Generation:**

* Most initial objects allocated in ‘Eden space’.
* Young generation also has two ‘survivor’ spaces.
* Objects that survive a GC get moved to survivor space.
* Only one survivor space in use at a time.
* Objects copied between survivor spaces.

All new objects are created here. Once it’s filled up the GC kick’s in. this type GC is called minor GC. Once this objects that won’t garbage collected (long lived objects), they are moved to Older Generation.



**Older/Tenured Generation:** Long lived objects are stored here. When this full major GC collects objects which are not useful any more.

**PermGen:**

PermGen (Permanent Generation) is a special heap space separated from the main memory heap.

The JVM keeps track of loaded class metadata in the PermGen. Additionally, the JVM stores all the static content in this memory section. This includes all the static methods, primitive variables, and references to the static objects.

Furthermore, it contains data about bytecode, names and JIT information. Before Java 7, the String Pool was also part of this memory. The disadvantages of the fixed pool size.

However, we can change the default size with the JVM options:

-XX:PermSize=[size] is the initial or minimum size of the PermGen space

-XX:MaxPermSize=[size] is the maximum size

Most importantly, Oracle completely removed this memory space in JDK 8 release.

With its limited memory size, PermGen is involved in generating the famous OutOfMemoryError. Simply put, the class loaders aren’t garbage collected properly and, as a result, generated a memory leak.

**Meta space in JAVA 8:**

Metaspace is a new memory space – starting from the Java 8 version; **it has replaced the older PermGen memory space**. The most significant difference is how it handles the memory allocation.

As a result, **this native memory region grows automatically by default**.

Additionally, the garbage collection process also gains some benefits from this change. The garbage collector now automatically triggers cleaning of the dead classes once the class metadata usage reaches its maximum metaspace size.

Therefore,**with this improvement, JVM reduces the chance to get the *OutOfMemory* error**.

Despite all of this improvements, we still need to monitor and [tune up](https://www.baeldung.com/jvm-parameters) the metaspace to avoid memory leaks.

Jstat –gccapacity pid

Jstat –gcutil pid

Jstat –gc pid

Jstat (JVM Statistics)

Jinfo (JVM configuration info)

Jmap (Java memory map)

Jstack (stack traces)

Jconsole (JMX MBeans)

Jcmd (combines jmap, jstack, jstat and jinfo into one) Command line equivalent of Java mission control (jmc.exe).

Commands:

Jcmd (displays pid of application)

jcmd [pid] help

jcmd 16896 Thread.print (thread info)

jcmd 16896 GC.heap\_info (show GC info) or jstat -gcutil 16896

jcmd 16896 GC.class\_histogram | less (class info)

jcmd [pid] VM.log what=gc output=[file-path]

gceasy.io, gcviewer to view above generated file.

**Continues Recording:**

jcmd 16896 JFR.start settings=default name=record1 maxage=4h

jcmd 16896 JFR.dump name=record1 filename=C:\Users\mupalla\Documents\record1.jfr

**Stack:**

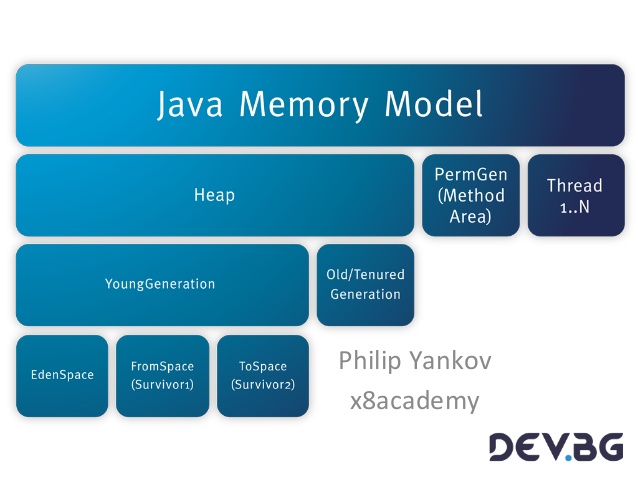
Local primitive variables, local objects, method parameters.

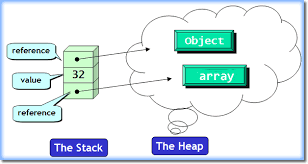
**Native Memory:**

Other language specific details.

**PC Register:**

Sequence of instruction which is executed next. One PC register per thread.





Primitive data types and object references are stored in the stack. Objects are stored in the heap.

**GC:**

**Marking:** what are the objects which are not used.

**Normal Deletion:** unused objects are removed.

**Deletion + Compacting:** unused objects are removed and grouping memory together.