



# Java Programming

1-2

JVM Memory Structure



# Objectives

This lesson covers the following topics:

- Introduce the Java Heap Memory
- Garbage collection
- Analyze the Memory allocation in JVM



# Java Heap Memory

- As we learned in a previous lesson the heap is where object data is stored.
- This area is then managed by the garbage collector at startup of the JVM.
- When the heap becomes full, garbage is collected.
- During the garbage collection, objects that are no longer used are cleared, making space for new objects.

# Java Heap Memory

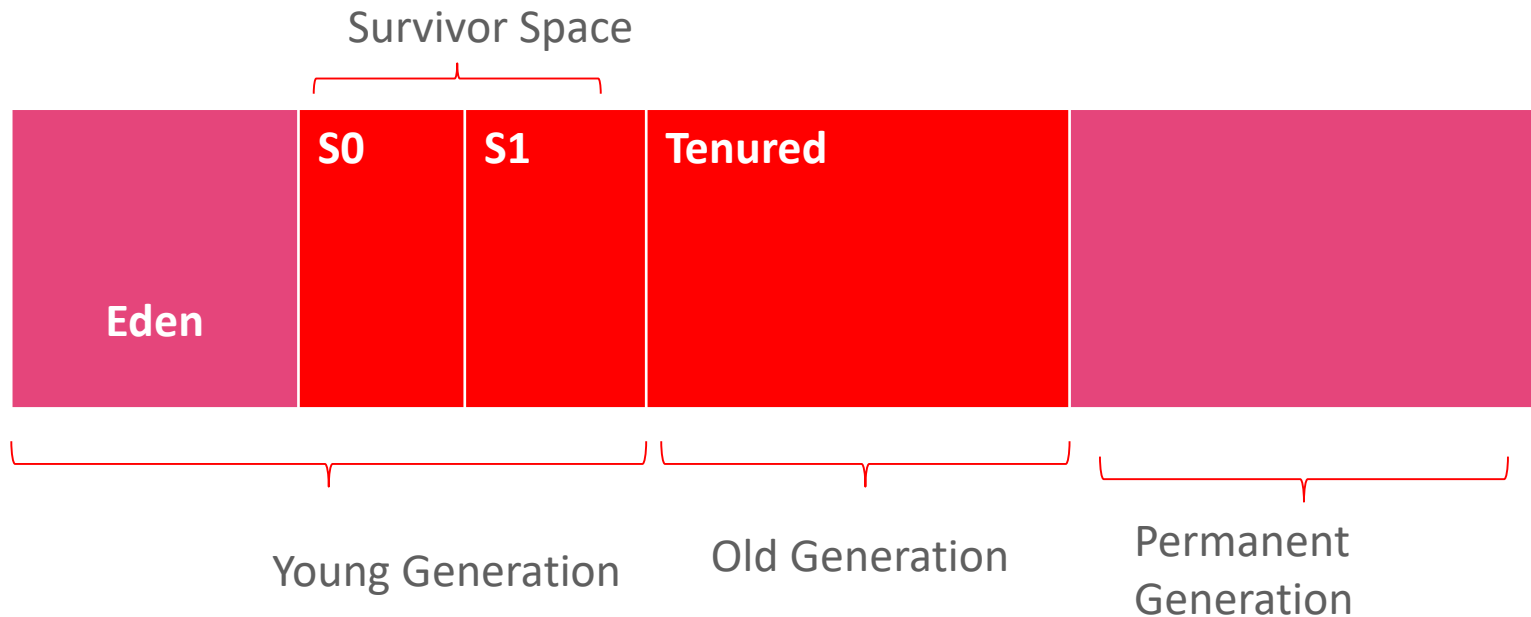
- The JVM Memory Model and Java Memory Management are very important features to understand how Java Garbage Collection works and the impact to the performance of your Java Application.
- The heap is sometimes divided into two areas (or generations) called the nursery (or young space) and the old space.
- The nursery is a part of the heap reserved for allocation of new objects.

# Java Heap Memory

- When the nursery becomes full, garbage is collected by running a special young collection process, where all objects that have lived long enough in the young space are promoted to the old space, freeing up the nursery for more object allocation.
- When the old space becomes full, garbage is collected there, a process called an old collection is initiated.

# Java Heap Memory – JDK7

Java memory in *HotSpot JVM* is managed in *generations*. Java Heap in *HotSpot JVM* is allocated into generational spaces.



# Java Heap Memory – JDK7

- In JDK 7, JVM memory is divided into separate parts. At a broad level, JVM Heap memory is physically divided into two parts , Young Generation and Old Generation.
- Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application.
- Perm Gen data is populated by JVM at runtime based on the classes used by the application.
- Perm Gen objects are garbage collected in a full garbage collection.



# Java Heap Memory – JDK7

- The reason we have two spaces is that garbage collected heaps should be structured in such a way to allow short-lived objects to be quickly collected, and long-lived objects to be separated from short-lived objects.
- These hypotheses are called the **weak generational hypothesis**.
  - Most objects soon become unreachable.
  - References from old objects to young objects only exist in small numbers.

# Young space

- Young Space is divide into three parts – **Eden Memory** and two **Survivor Memory** spaces.
- Most of the newly created objects are located in the Eden Memory space
- When Eden space is filled with objects, Minor Garbage Collection is performed and all the survivor objects are moved to one of the survivor spaces
- Minor Garbage Collection also checks the survivor objects and moves them to the other survivor space.

# Young space

- At any time, one of the survivor spaces is always empty
- Objects that have survived many cycles of Garbage Collection are moved to the old generation memory space.
- Usually it is done by setting a threshold for the age of the nursery objects before they become eligible to be promoted to old generation
- The majority of newly created objects are located in the Eden space.

# Young space

- There are 3 spaces in total, two of which are Survivor spaces.
- The order of execution process of each space is as below:
  - After one GC in the Eden space, the surviving objects are moved to one of the Survivor spaces.
  - After a GC in the Eden space, the objects are piled up into the Survivor space, where other surviving objects already exist.
  - Once a Survivor space is full, surviving objects are moved to the other Survivor space.

# Young space

- The objects that survive these steps that have been repeated a number of times are moved to the old generation.

# Old Space

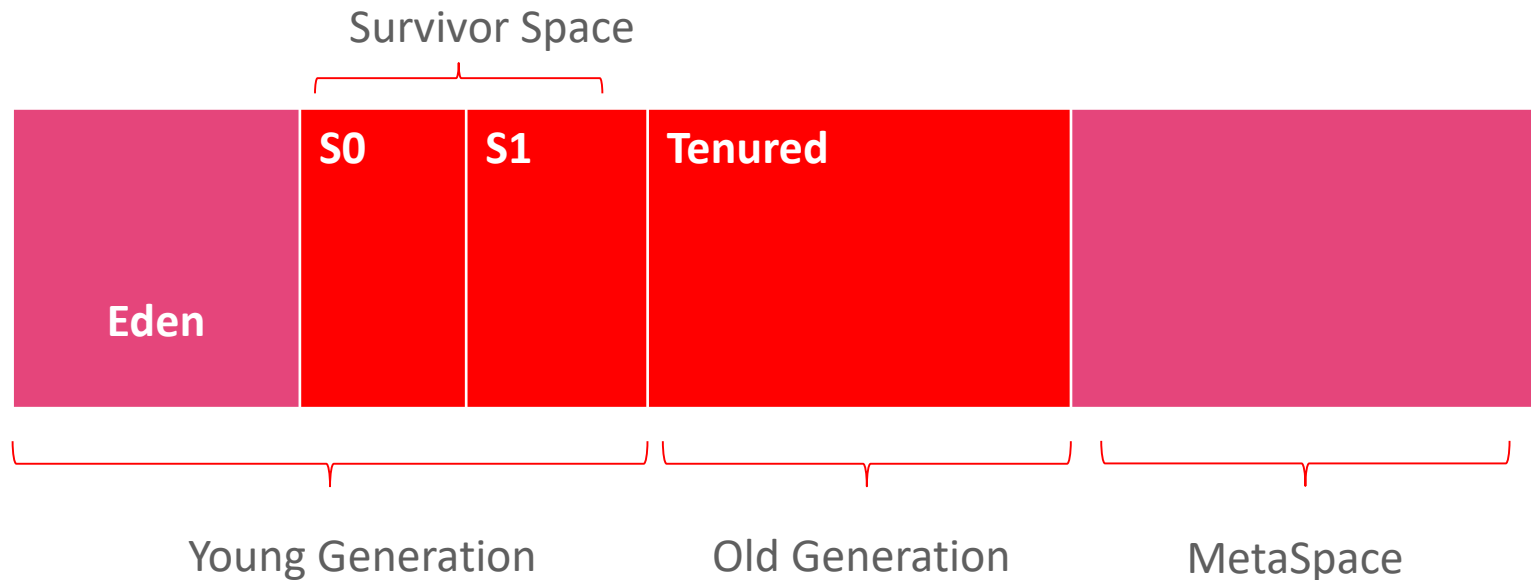
- When the old generation becomes full, garbage is collected there and the old collection process is performed.
- Old generation memory contains the objects that are long lived and have survived after many rounds of Minor Garbage Collection.
- Usually garbage collection is performed in Old generation memory when it's full.

# Old Space

- Old generation garbage collection is called as **Major Garbage Collection** and usually takes a longer time.
- Most objects are temporary and short lived.
- A young collection is designed to be swift at finding newly allocated objects that are still alive and moving them away from the Young space.
- Typically, a young collection frees a given amount of memory much faster than an old collection.

# Java Heap Memory – JDK8

- Java memory is managed in generations in HotSpot JVM.
- Java Heap is allocated into generational spaces in HotSpot JVM.





# Java Heap Memory – JDK8

- In JDK 8. the Java Heap is still divided into two parts: Young space and Old Space.
- The Permanent Generation (Perm Gen) space has been removed from the JDK 8 release.
- The PermSize and MaxPermSize options are ignored and a warning is issued if they are present.

# Java Heap Memory – JDK8

- By default, class metadata allocation is only limited by the amount of available native memory.
- Use the new flag `MaxMetaspaceSize` to limit the amount of native memory used for the class metadata.
- With the introduction of `MetaSpace`, JVM will not run out of `PermGen` space, Java Applications may consume excessive Native Memory.

# JDK 8 MetaSpace Example

- In order to better understand the runtime behavior of the new Metaspace memory space in JDK 8 , we will create a class metadata leaking Java program.

```
PS C:\test\mem> java -classpath ".;meta" -XX:MaxMetaspaceSize=11m MetaspaceOOM
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
Hello world 0
Hello world 1
Hello world 2
Hello world 3
```

```
Hello world 521
Hello world 522
Hello world 523
Hello world 524
Hello world 525
Hello world 526
Exception in thread "main" java.lang.OutOfMemoryError: Metaspace
```

# JDK 8 MetaSpaceOOM Example

```
File sourceFile = new File("Hello" + i + ".java");  
FileWriter writer = new FileWriter(sourceFile);  
writer.write("public class Hello" + i + " { public void sayHello()  
{ System.out.println(\"Hello world \" + i + "\");  
} }");
```

- In this example is we are going to create multiple java source files which start from the number zero and increase to 1000. We will create a java source code file named from Hello0.java to Hello1000.java

# JDK 8 MetaSpace Example

- To simulate the depletion of the Metaspace we run the Java program using JDK 1.8 to by setting the MaxMetaspaceSize Value at 11 MB.
- The MetaspaceOOM program will generate the Hello java file and compile into the Hello class.
- All of these classes will be loaded and class metadata will be stored in the Metaspace area.
- In this configuration, after loading 526 Hello classes, the metaspace will be full and an OutOfMemoryError Exception is thrown from the JVM.

# JDK 8 MetaSpaceOOM Example

```
// Compile the Hello Java file
```

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
```

```
StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null, null);
```

```
Iterable <? extends File> path= Collections.singletonList(new File("C:\\test\\mem\\meta"));
```

```
fileManager.setLocation(StandardLocation.CLASS_OUTPUT, path);
```

```
compiler.getTask(null, fileManager, null, null, null, fileManager.getJavaFileObjectsFromFiles(Collections.singletonList(sourceFile))).
```

```
call();
```

```
fileManager.close();
```

# JDK 8 MetaSpaceOOM Example

- `JavaCompiler` is an interface to invoke Java Programming language compilers from programs.
- A compiler tool has an associated standards file manager `StandardJavaFileManager`, a common way to obtain an interface of this class is using `getStandardFileManager`.
- Before the call to `method()`, we have to obtain a `JavaCompiler.CompilationTask` instance which requires the information for `JavaFileManager` and `JavaFileObject`.

# JDK 8 MetaSpace Example

```
PS C:\test\mem> jmap -heap 232540
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
Attaching to process ID 232540, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.121-b13

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
  MinHeapFreeRatio      = 0
  MaxHeapFreeRatio      = 100
  MaxHeapSize            = 4217372672 (4022.0MB)
  NewSize                = 88080384 (84.0MB)
  MaxNewSize             = 1405616128 (1340.5MB)
  OldSize                = 176160768 (168.0MB)
  NewRatio               = 2
  SurvivorRatio          = 8
  MetaspaceSize          = 11534336 (11.0MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize       = 11534336 (11.0MB)
  G1HeapRegionSize       = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
  capacity = 203423744 (194.0MB)
  used     = 154996752 (147.81642150878906MB)
  free     = 48426992 (46.18357849121094MB)
  76.19403170556137% used
From Space:
  capacity = 87556096 (83.5MB)
  used     = 32000704 (30.51824951171875MB)
  free     = 55555392 (52.98175048828125MB)
  36.54880181044162% used
To Space:
  capacity = 97517568 (93.0MB)
  used     = 0 (0.0MB)
  free     = 97517568 (93.0MB)
  0.0% used
PS Old Generation
  capacity = 299892736 (286.0MB)
  used     = 173258008 (165.2317123413086MB)
  free     = 126634728 (120.7682876586914MB)
  57.773325993464546% used
```

Jmap to check the size of MetaspaceSize

We can use the jmap utility to monitor the usage of the Metaspace area.

In this example, we can see the MaxspaceSize reach 11 MB.



# JDK 8 Garbage Collection

- Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.
- An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object.
- An unused object, or unreferenced object, is no longer referenced by any part of your program.

# JDK 8 Garbage Collection

- So the memory used by an unreferenced object can be reclaimed.
- Java can not explicitly free an object in Java source code and Java byte code.
- The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application.
- The Java virtual machine uses a garbage collector to manage the heap.

# JDK 8 Garbage Collection

- The essence of Java's garbage collection is that rather than requiring the programmer to understand the precise lifetime of every object in the system, the JVM should keep track of objects on the programmers behalf and automatically get rid of objects that are no longer required.
- The automatically reclaimed memory can then be wiped and reused.

# JDK 8 Garbage Collection

- There are two fundamental rules of garbage collection that all implementations are subject to:
  - Garbage Collection must collect all garbage.
  - No live object can ever be collected.
- The Java HotSpot VM offers three different types of collectors, in this course we will focus on:
  - Serial Collector
  - Parallel Collector

# JDK 8 Garbage Collection – Serial Collector

- The Serial Collector uses a single thread to perform all garbage collection work, which makes it relatively efficient because there is no communication overhead between threads.
- The serial collector is selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option `-XX:+UseSerialGC`.

# JDK 8 Garbage Collection – Serial Collector

- With the Serial Collector, both young and old collections are done serially by using a single CPU, in a stop-the-world fashion.
- The application execution is halted while collection is taking place.
- Most of time, the client-style machine runs the application using a serial collector.

# JDK 8 Garbage Collection – Parallel Collector

- The parallel collector (also known as the *throughput collector*) performs minor collections in parallel, which can significantly reduce garbage collection overhead.
- The parallel collector is selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option `-XX:+UseParallelGC`.
- The parallel collector was developed in order to take advantage of a available CPU rather than leaving most of them idle.

# JDK 8 Garbage Collection – Parallel Collector

```
C:\test\mem>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+PrintCommandLineFlag  
s -XX:+PrintTenuringDistribution TestMemory  
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en  
-XX:InitialHeapSize=20971520 -XX:MaxHeapSize=20971520 -XX:MaxNewSize=10485760 -XX:NewSize=10485760 -XX:+PrintCommandLine  
Flags -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintTenuringDistribution -XX:SurvivorRatio=8 -XX:+UseCompressedClassPointer  
s -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC  
[GC (Allocation Failure)
```

By default, in this case, JVM uses the ParallelGC.

After set the option `-Xms20M` so the initial size reserved for the heap by JVM is -  
`XX:InitialHeapSize=20971520`

After set the option `-Xmx20M` so the maximum size reserved for the heap by JVM is -  
`XX:MaxHeapSize=20971520`

`-XX:MaxNewSize=10485760` the maximum Young generation size is 10M

`-XX:NewSize=10485760` the minimum Young generation size is 10M

`-XX:SurvivorRatio=8` Ratio of Eden area and Survivor area

`-XX:+UseParallel` garbage collector used.



# JDK 8 Garbage Collection

- When developing an application you are primarily looking for two things from Garbage Collection Log output.
- **Is the GC happening very frequently?**
  - In an application, ideally Garbage Collection cycles should not be frequent – at least every few minutes (especially the Major Collection or full Garbage Collection).
  - If you see Garbage Collection kicking in every few seconds, you should investigate the possible reasons why.

# JDK 8 Garbage Collection

- **Is each GC cycle taking a long time?**
  - Ideally minor Garbage Collections should take few milliseconds and Full Garbage Collections should take about a second or less.
  - If you see Garbage Collection taking longer than this, you should investigate the possible reasons why.

# JDK 8 Garbage Collection

- *Ergonomics* is the process by which the Java Virtual Machine (JVM) and garbage collection tuning, such as behavior-based tuning, improve application performance.
- The JVM provides platform-dependent default selections for the garbage collector, heap size, and runtime compiler.

# Diagnosing a Garbage Collection output

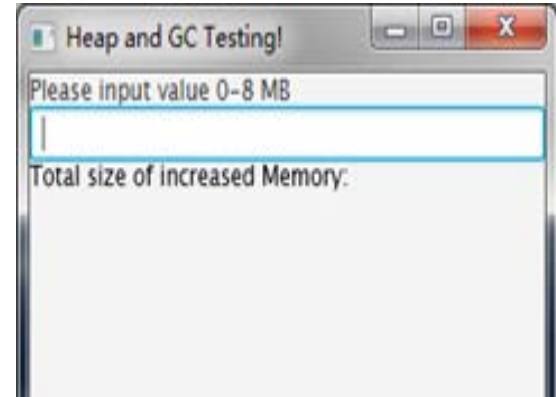
- We can generate the verbosegc output by using the following option during the runtime of the Java application:
  - `-verbosegc -XX:+ PrintGCDetails`
- The minor collection output for these options produce output of the form:
  - `[GC [<collector>: <starting occupancy1> -> <ending occupancy1>, <pause time1> secs] <starting occupancy3> -> <ending occupancy3>, <pause time3> secs]`

# Diagnosing a Garbage Collection output

- <collector> is an internal name for the collector used in the minor collection.
- <starting occupancy1> is the occupancy of the young generation before the collection.
- <ending occupancy1> is the occupancy of the young generation after the collection.
- <pause time1> is the pause time in seconds for the minor collection.

# Monitoring Garbage Collection JDK 8 Java Heap (Example)

There are 3 Young Garbage Collection that happened after we ran the Testing application.



```
C:\test\mem>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+PrintTenuringDistribution -XX:+HeapDumpOn
tMemory
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 8188K->1004K(9216K)] 8188K->1764K(19456K), 0.0039729 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9196K->1008K(9216K)] 9956K->3055K(19456K), 0.0037699 secs] [Times: user=0.05 sys=0.02, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9200K->1016K(9216K)] 11247K->3825K(19456K), 0.0081551 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

# Monitoring Garbage Collection

## JDK 8 Java Heap (Example)

- “Allocation Failure” is the cause for minor GC.
- We see that the current tenuring threshold is 7 and that its maximum value is 15.
- PSYoungGen – the name of the garbage collector being used, representing a parallel mark-copy stop-the-world garbage collector used to clean the Young generation.

```
C:\test\mem>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+PrintTenuringDistribution -XX:+HeapDumpOn
tMemory
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 8188K->1004K(9216K)] 8188K->1764K(19456K), 0.0039729 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9196K->1008K(9216K)] 9956K->3055K(19456K), 0.0037699 secs] [Times: user=0.05 sys=0.02, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9200K->1016K(9216K)] 11247K->3825K(19456K), 0.0081551 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

# Monitoring Garbage Collection

## JDK 8 Java Heap (Example)

- 8188K->1004K - usage of the Young Generation before and after collection (9216K) - Total size of the Young Generation
- 8188K->1764K - Total heap usage before and after collection

```
C:\test\mem>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+PrintTenuringDistribution -XX:+HeapDumpOn
tMemory
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 8188K->1004K(9216K)] 8188K->1764K(19456K), 0.0039729 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9196K->1008K(9216K)] 9956K->3055K(19456K), 0.0037699 secs] [Times: user=0.05 sys=0.02, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9200K->1016K(9216K)] 11247K->3825K(19456K), 0.0081551 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```



# Monitoring Garbage Collection JDK 8 Java Heap (Example)

- (19456K) - Total available heap
- 0.0039729 secs - Pause time of the GC event in seconds

```
C:\test\mem>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+PrintTenuringDistribution -XX:+HeapDumpOn
tMemory
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 8188K->1004K(9216K)] 8188K->1764K(19456K), 0.0039729 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9196K->1008K(9216K)] 9956K->3055K(19456K), 0.0037699 secs] [Times: user=0.05 sys=0.02, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9200K->1016K(9216K)] 11247K->3825K(19456K), 0.0081551 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

# GC: Parallel Scavenge

- Generational, stop-the-world, parallel
- Collects young generation only
- Available since JDK™ 1.4.2 software
- First parallel Garbage Collection in Java HotSpot performance engine
- Used in many applications
- Majority of Garbage Collection time is spent on young generation
  - `XX:+UseParallelGC`.

```
C:\test\mem>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+PrintTenuringDistribution -XX:+HeapDumpOn
tMemory
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 3188K->1004K(9216K)] 8188K->1764K(19456K), 0.0039729 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9196K->1008K(9216K)] 9956K->3055K(19456K), 0.0037699 secs] [Times: user=0.05 sys=0.02, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9200K->1016K(9216K)] 11247K->3825K(19456K), 0.0081551 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

# GC: Parallel Scavenge

- In the other output from the older Oracle JVM version, young generation uses the DefNew garbage collector name – the name stands for the single-threaded mark-copy stop-the-world garbage collector used to clean Young generation.

```
C:\test\mem>java -verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+PrintTenuringDistribution -XX:+HeapDumpOn
tMemory
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[P5YoungGen: 8188K->1004K(9216K)] 8188K->1764K(19456K), 0.0039729 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[P5YoungGen: 9196K->1008K(9216K)] 9956K->3055K(19456K), 0.0037699 secs] [Times: user=0.05 sys=0.02, real=0.00 secs]
[GC (Allocation Failure)
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[P5YoungGen: 9200K->1016K(9216K)] 11247K->3825K(19456K), 0.0081551 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

# JDK 8 jstat monitor

From the jstat tool, we can see when we start the application, 3 Young Generation garbage collection events happened.

S0C: S0 capacity : 1024KB  
S1C: S1 capacity: 1024KB  
S0U: S0 Utility : 0  
S1U: S1 Utility: 1016KB  
EC: 8K  
EU: 5K  
OC: 10240K  
OU: 2902K  
YGC: 3  
FGCT: 0

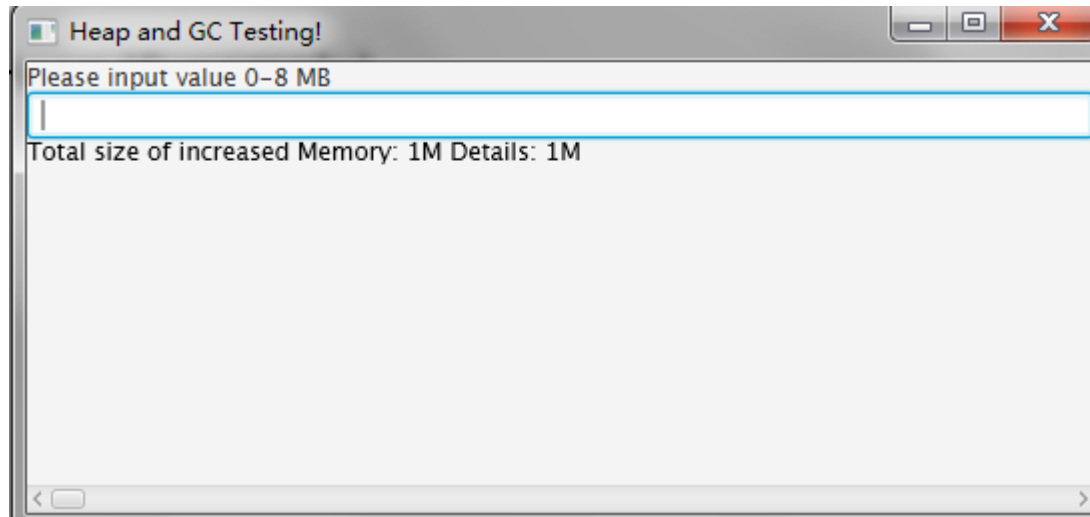
Jstat is a monitoring tool in HotSpot JVM.  
Jstat does not provide only the GC operation information display.

It also provides class loader operation information or Just-in-Time compiler operation information.

```
PS C:\test\mem> jstat -gc 241976 10000
Picked up JAVA_TOOL_OPTIONS: -Duser.language=en
S0C   S1C   S0U   S1U   EC    EU    OC    OU    MC    MU    CCSC  CCSU  YGC   YGCT  FGC   FGCT  GCT
1024.0 1024.0 0.0   1016.0 8192.0 5545.9 10240.0 2902.6 13952.0 13122.5 2176.0 1979.6 3     0.016 0     0.000 0.016
```

# JDK 8 Java Heap

Step 1 add 1M size memory:



- We can ask the application to create 1M Normal java Object.
- From the jstat tool, we can check the output

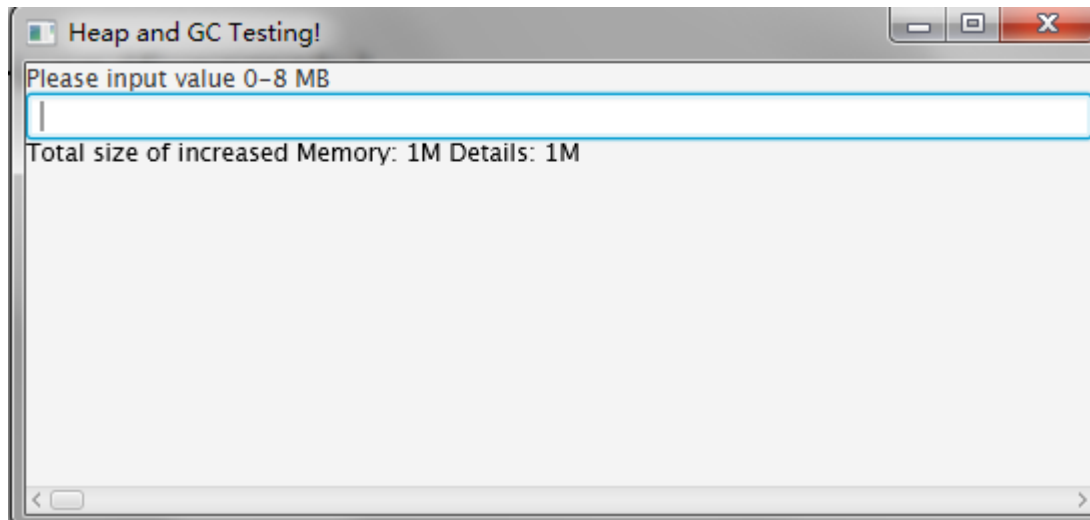
# JDK 8 Java Heap

- We can see the new 1M memory allocated in Young Generation, Eden space increase from 6087 to 8173. no GC.
- The output from the application might be different depending on the native computer configuration.
- After a request for 1M heap memory from the JVM, the 1M data will reside in the Eden space of Young generation and no Minor GC will be triggered.

1024.0	1024.0	0.0	1008.1	8192.0	6087.2	10240.0	2840.9	13696.0	13049.0	2176.0	1970.7	3	0.010	0	0.000	0.010
1024.0	1024.0	0.0	1008.1	8192.0	8173.5	10240.0	2840.9	13696.0	13049.0	2176.0	1970.7	3	0.010	0	0.000	0.010

# JDK 8 Java Heap

Step 2 add 1M 1M



- We can ask the application to create 1M Normal java Object. From the jstat tool, we can check.

# JDK 8 Java Heap

1024.0	1024.0	0.0	1008.1	8192.0	6087.2	10240.0	2840.9	13696.0	13049.0	2176.0	1970.7	3	0.010	0	0.000	0.010
1024.0	1024.0	0.0	1008.1	8192.0	8173.5	10240.0	2840.9	13696.0	13049.0	2176.0	1970.7	3	0.010	0	0.000	0.010
S0C	S1C	SOU	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	GCT
1024.0	1024.0	0.0	0.0	8192.0	1880.4	10240.0	8090.4	15360.0	14358.6	2304.0	2106.5	4	0.030	1	0.070	0.101

- We can see the new 1M memory triggers a young GC as Eden is full, it also triggers a Full GC, as we need to move the object from Young to Old generation.

```
Desired survivor size 1048576 bytes, new threshold 7 (max 15)
[PSYoungGen: 9200K->1008K(9216K)] 12040K->8380K(19456K), 0.0204298 secs] [Times: user=0.03 sys=0.02, real=0.02 secs]
[Full GC (Ergonomics) [PSYoungGen: 1008K->0K(9216K)] [ParOldGen: 7372K->8090K(19456K)] 8380K->8090K(19456K), [Metaspace: 14368K->912K], 0.0702412 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]
```

- From the output of jstat utility, there are 4 Young garbage collections and 1 Full garbage collection happened when 1M of memory allocated.



# GC: ParOldGen

```
Desired survivor size 1048576 bytes, new threshold 7 (max 15)  
[PSYoungGen: 9200K->1008K(9216K)] 12040K->8380K(19456K), 0.0204298 secs] [Times: user=0.03 sys=0.02, real=0.02 secs]  
[Full GC (Ergonomics) [PSYoungGen: 1008K->0K(9216K)] [ParOldGen: 7372K->8090K(10240K)] 8380K->8090K(19456K), [Metaspace: 14368K  
912K)], 0.0702412 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]
```

- Stop-the-world, parallel, sliding compaction
- Full GC—collects entire heap
- Paired with Parallel Scavenge
- -XX:+UseParallelOldGC

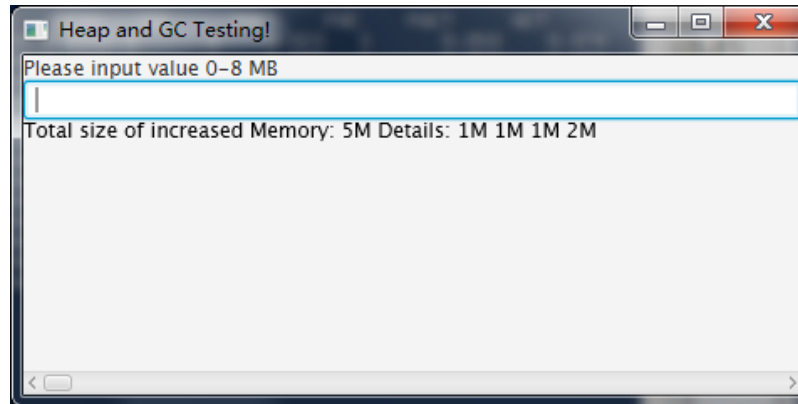
# GC: ParOldGen

```
Desired survivor size 1048576 bytes, new threshold 7 (max 15)  
[PSYoungGen: 9200K->1008K(9216K)] 12040K->8380K(19456K), 0.0204298 secs] [Times: user=0.03 sys=0.02, real=0.02 secs]  
[Full GC (Ergonomics) [PSYoungGen: 1008K->0K(9216K)] [ParOldGen: 7372K->8090K(10240K)] 8380K->8090K(19456K), [Metaspace: 14368K  
912K)], 0.0702412 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]
```

- A programmer can specify to use the ParOldGen GC by providing the option of `-XX:+UseParallelOldGC`
- Full GC (Ergonomics) indicates the choice for the garbage collector, heap size, and HotSpot virtual machine (client or server) are automatically chosen based on the platform and operating system on which the application is running.

# JDK 8 Java Heap

- Example Step 3 add 1M 1M 1M 2M



- We can ask the application to create 4M Normal java Object.
- From the jstat tool, we can check.

1024.0	1024.0	0.0	0.0	8192.0	2010.2	10240.0	7233.6	15104.0	14286.7	2304.0	2106.2	4	0.034	1	0.028	0.062
1024.0	1024.0	0.0	0.0	8192.0	3997.8	10240.0	7233.6	15104.0	14286.7	2304.0	2106.2	4	0.034	1	0.028	0.062
1024.0	1024.0	0.0	0.0	8192.0	5931.3	10240.0	7233.6	15104.0	14286.7	2304.0	2106.2	4	0.034	1	0.028	0.062

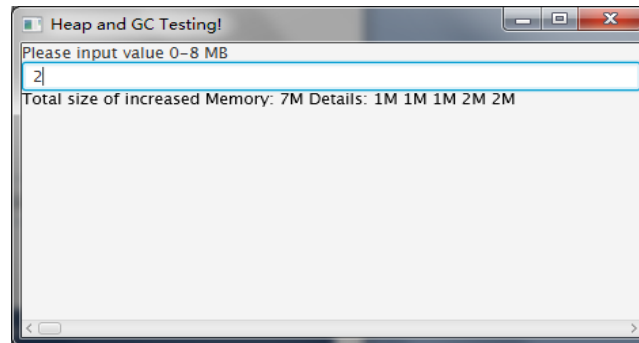
# JDK 8 Java Heap

1024.0	1024.0	0.0	0.0	8192.0	2010.2	10240.0	7233.6	15104.0	14286.7	2304.0	2106.2	4	0.034	1	0.028	0.062
1024.0	1024.0	0.0	0.0	8192.0	3997.8	10240.0	7233.6	15104.0	14286.7	2304.0	2106.2	4	0.034	1	0.028	0.062
1024.0	1024.0	0.0	0.0	8192.0	5931.3	10240.0	7233.6	15104.0	14286.7	2304.0	2106.2	4	0.034	1	0.028	0.062

- We can see the change of EU column after new 1M and 1M memory generated
- So the Eden space increase from 2010.2k to 3997.8k and 5931.3K after we request 1M 1M and 1M size of the memory heap.

# JDK 8 Java Heap

- Example Step 3 add 1M 1M 1M 2M



- Continue the process, until we reach the OOM of Heap space

```
Exception in thread "JavaFX Application Thread" java.lang.OutOfMemoryError: Java heap space
    at TestMemory.increaseMemory(TestMemory.java:96)
    at TestMemory.lambda$start$0(TestMemory.java:50)
    at TestMemory$$Lambda$79/2092292816.handle(Unknown Source)
```

- When asked for more heap memory from The JVM, when no more Heap is available, then the JVM would throw a Out of Memory exception.

# Summary

In this lesson, you should have learned:

- Introduce the Java Heap Memory
- Garbage collection
- Analyze the Memory allocation in JVM



