What is The Heap?

When the Java Virtual Machine is started, it is allocated some memory by the Operating System. Some of this memory is referred to as *the heap*. All Java objects created reside in the heap. When the heap becomes full, a process called garbage collection occurs - objects that are no longer used are cleared in order to make space for new objects.

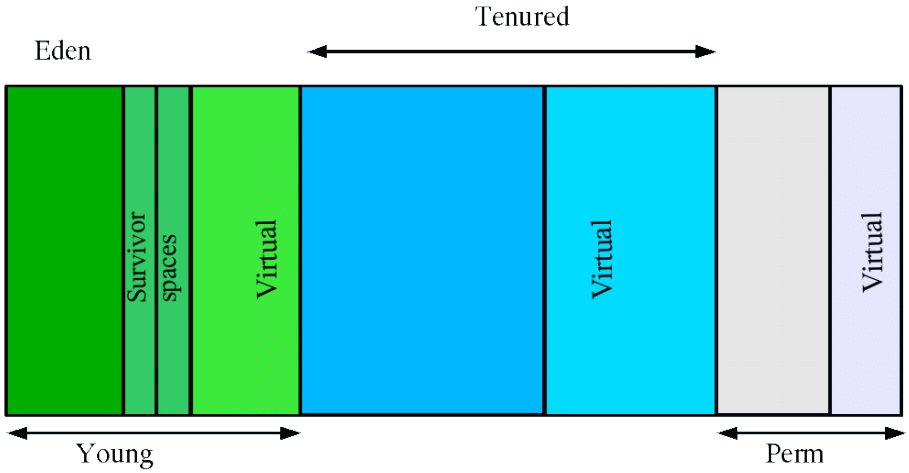
What does the `new` operator do?

The `new` operator instantiates a class by allocating memory for a new object on the heap and returning a reference to that memory. The new operator also invokes the object constructor.

What is Garbage Collection, and how does it work?

The JVM uses a form of garbage collector called a tracing collector, which essentially operates by pausing the world around it, marking all root objects (objects referenced directly by running threads), and following their references, marking each object it sees along the way.

To understand garbage collection, we must first understand how the memory allocated to the JVM is divided. Java implements something called a generational garbage collector based upon the generational hypothesis assumption - majority of objects that are created are quickly discarded, and objects that are not quickly collected are likely to be around for a while. Keeping this in mind, the diagram below shows different generations of the allocated memory:



- Young Generation – This is where all objects start out. This generation has two sub-spaces:
  - Eden Space: Allocating new objects using `new` operator almost always happens in Eden space. In the Eden Space, the GC performs a process called Minor GC, which is an optimized garbage collection process. During the first Minor GC, the JVM copies all objects in use to one of the survivor spaces, and then clears the Eden Space. In subsequent Minor GCs, the GC examines the Eden Space along with one of the Survivor Spaces; live objects from both are copied to the second Survivor Space.

    - Survivor Spaces: Objects that survive the Minor GC end up here. Only one of these is in use at a time - One is designated as *empty*, and the other as *live*, alternating with every GC cycle.

- Tenured Generation – this space holds older objects with longer lifetimes. The JVM determines which objects to promote by checking how many times an object has jumped back and forth between Survivor Spaces – once a given object jumps back and forth too many times (configurable, 8 by default), it is promoted to tenured space.

  When the Tenured Generation is full, a Full GC is performed. This is a much costlier process as compared to the Minor GC.

- Permanent Generation – This generation is not part of The Heap (non-heap memory). It holds data required by the virtual machine to describe objects that do not have an equivalence at the Java language level. For example, objects describing classes and methods are stored in the permanent generation.

What causes an `Out of Memory (OOM)` Error?

`java.lang.OutOfMemoryError` is an exception thrown when:

- The Java Virtual Machine could not allocate an object because it is out of memory, no more memory could be made available by the garbage collector and the heap space could not be expanded.
- May be thrown when there is insufficient native memory to support the loading of a Java class.
- Rarely, may be thrown when an excessive amount of time is being spent doing garbage collection and little memory is being freed.

How can OOMs be diagnosed?

Java OOM errors are of the following types:

1. `java.lang.OutOfMemoryError:Java heap space`
   - Indicates object could not be allocated in the Java heap. Could be a result of:
     - Configuration issue, where the specified heap size is insufficient for the application.
     - Unintentionally holding references to objects, preventing the garbage collector from cleaning them up.
     - Excessive usage of finalizers. If a class has a finalize method, then objects of that type do not have their space reclaimed at garbage collection time. Instead, after garbage collection, the objects are queued for finalization, which occurs later. If the finalizer thread cannot keep up with the finalization queue, then the Java heap could fill up and an OOM could be thrown.

2. `java.lang.OutOfMemoryError: GC Overhead limit exceeded`
   If the GC spends >98% of its time on garbage collection, and frees only <2% of the heap, this implies that the Java program is making very slow progress. If this happened over 5 consecutive GC cycles, an OOM error is thrown.

3. `java.lang.OutOfMemoryError:Requested array size exceeds VM limit`
   This error indicates that the application (or APIs used by that application) attempted to allocate an array that is larger than the heap size. In most cases, the problem is either a configuration issue or a bug that results when an application attempts to allocate a massive array.

4. `java.lang.OutOfMemoryError:Metaspace`
   Java class metadata (the VM's internal presentation of Java class) is allocated in native memory referred to as metaspace. When the amount of native memory needed for a class metadata exceeds a threshold value (`MaxMetaSpaceSize)`, an OOM error is thrown. The main cause for these errors is either too many classes or too big classes.

5. `java.lang.OutOfMemoryError:request <size> bytes for <reason>. Out of swap space?`
   Occurs when an allocation from the native heap failed and the native heap might be close to exhaustion. In most cases, the <reason> is the name of the source module that's reporting an allocation failure.
   In some cases, this error might occur due to the OS being configured with insufficient swap space or if another process on the system is consuming all available memory resources.

6. `java.lang.OutOfMemoryError:<reason><stack_trace> (Native Method)`
   If a stack trace is printed with a native method on top of the stack, then this is an indication that a native method has encountered an allocation failure.

How can OOMs be fixed?

1. Java Heap Space
   i. In some cases, the amount of heap space allocated is simply not enough. In these cases, just allocate more heap space using -Xmx (e.g. `java -Xmx1073741824 com.mycompany.MyClass`).
   ii. If the Java Application contains a memory leak, allocating more heap space will just prolong the OOM error. This will also increase the length of GC pauses which slows down the application. In order to truly solve the problem, one must perform a Heap Dump of the JVM and analyze it to understand which objects occupy large portions of heap and where these objects are being allocated in source code. With this information, you can modify your code so that it works with your heap size.
2. GC Overhead limit
   i. In some cases allocating more heap space should do the trick.
   ii. In most cases, however, perform a heap dump analysis to target the root problem.
3. Requested array size exceed VM limit
   i. Check your code base to see whether you really need arrays that large. If not, reduce the size of the arrays or divide the array into smaller bulks and load the data you need to work with in batches fitting into your platform limit.

ii. If you really need to use large data sets, you can use use utilities like `sun.misc.Unsafe` that allow you to allocate memory directly like in C.

4. Metaspace
   i. Allocate more metaspace by using `-XX:MaxMetaspaceSize` or remove the limit altogether (can introduce heavy swapping and/or reach native allocation failures instead).
5. Out of Swap Space?
   i. Increase swap size.
   ii. If your application is deployed next to a "noisy neighbor" with whom the JVM needs to compete for resources, you should isolate the services to separate (virtual) machines.
   iii. Either upgrade the machine to contain more memory or optimize the application to reduce its memory footprint. A good way to start is by using memory dump analyzers to detect large allocations in memory.
6. Native Method
   i. Use native utilities of the OS to further diagnose the issue. See: https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr020.html#BABBHHIE.

Coding Practices

Here is a list of coding practices that can be followed in order to reduce the frequency of OOM errors:

Profiling

Profiling is a form of program analysis that directly measures various parameters like memory usage, time taken for program completion, frequency and duration of function calls etc. with the primary aim of optimizing programs.

Profiling can be very useful when writing new code – adding a few lines of code to measure time taken and memory usage can go a long way in helping to reduce the frequency of OOM errors found in production. If a memory leak can be pinpointed in the early stages of a code snippet, it is much easier to fix.

Java comes with some functions that make it very easy to profile your code:

- Time

```
long startTime = System.nanoTime();
methodToTime();
long endTime = System.nanoTime();
long duration = (endTime - startTime);
```

- Memory
  - `Runtime.getRuntime().freeMemory()` - Free Memory in the JVM
  - `Runtime.getRuntime().maxMemory()` - Max amount of memory the JVM will

attempt to use

- `Runtime.getRuntime().totalMemory()` - Total amount of memory in the JVM
- `Runtime.getRuntime().gc()` - Runs the garbage collector; run this at the end of a function to ensure that the GC is functioning properly and that there are no memory leaks in the code you have written.
- Memory usage = `totalMemory() - freeMemory()`

HashMap

A HashMap is a hash table implementation for the Map interface, and as such it defines the basic concepts of key and value: each value is related to a unique key, so if the key for a given key-value pair is already present in the HashMap, its current value is replaced. A HashMap requires that the equals() and hashcode() methods to be implemented correctly. If this is not done properly, repeated elements will be added to the HashMap, causing it to grow uncontrollably – this can quickly fill up the Heap and result in an OOM. Thus, it is essential to implement the equals and hashcode functions when using a HashMap.

HSSF, XSSF and SXSSF

HSSF is the POI Project's Java implementation of the Excel (1997-2007) file format, whereas the XXF is the Java Implementation of the Excel 2007 OOXML file format. Using both these APIs with large Excel files can prove to be very memory hungry and hence it is wiser to use SXSSF, a streaming extension of the XSSF. SXSSF achieves its low memory footprint by limiting access to the rows that are within a sliding window, while XSSF gives access to all rows in the document. Older rows that are no longer in the window become inaccessible, as they are written to the disk.

Along with the usage of SXSSF, a dramatic improvement in memory usage can be done by using a File instead of a Stream which buffers the whole file into memory. Instead of using `WorkbookFactory.create(inputStream);` you can use `WorkbookFactory.create(new File("yourfile.xlsx"));`.

It is important to note that SXSSF has some limitations (See: https://poi.apache.org/spreadsheet/).

JDBC

Trying to read a large amount of data from MySQL using Java using one query can easily place a large amount of data on The Heap, which can quickly lead to a OOM error. In order to avoid all the data being loaded into the heap at the same time, it is better to load smaller chunks of data, process it and then pull the next set. When using JDBC, this can be done in the following manner:

```
// make sure autocommit is off
conn.setAutoCommit(false);
Statement st = conn.createStatement();
st.setFetchSize(50); //define number of rows to pull at a time
ResultSet rs = st.executeQuery("SELECT * FROM mytable");
while (rs.next())
```

```
        System.out.print("a row was returned.");
    rs.close();
```

To pull rows one by one (Streaming mode), the following method can be used:

```
    stmt =
    conn.createStatement(java.sql.ResultSet.TYPE_FORWARD_ONLY,
                java.sql.ResultSet.CONCUR_READ_ONLY);
    stmt.setFetchSize(Integer.MIN_VALUE);
```

Setting the fetch size to Integer.MIN_VALUE tells the driver to stream the result set row-by-row. This method does have some issues; if using streaming results, you should process them as quickly as possible if you want to maintain concurrent access to the tables you are acting upon. i.e. you will have to read all rows in the result set (or close it) before you can fire another query.

SELECT * queries

When requesting only a single row, such as `SELECT * FROM a_table WHERE some_column = some_unique_value`, it is still generally advisable not to use `SELECT *` because the definition of the table may change over time. For example, a table may have a column added, removed, renamed or re-positioned relative to other columns. Such changes can result in SQL queries returning data that are never used or attempting to implicitly access data that does not exist.

Using `SELECT *` can also affect performance. If you were to use *, and it returned more columns than you actually needed, the server would often have to perform more expensive methods to retrieve your data than it otherwise might. Also, all fields might not be indexed, forcing a full table scan which is much less efficient.

Thus, it is advisable to explicitly list every column that is needed in the query to avoid performance and maintenance issues. Additionally, this also reduces the amount of memory that result sets will use in The Heap, thus reducing the chances of an OOM error from occurring.

String Concatenation

In string concatenation, it is normal to use the + operator for one concatenation:

```
        String x = "a" + "b";
```

This creates a StringBuilder object in the background to append the string “b” to “a”. But, what happens when this statement is placed in a loop? It creates multiple instances of the StringBuilder object that puts unnecessary pressure on the Heap and the GC. To avoid this, As a rule of thumb, use a StringBuilder rather than the + operator like so:

```
        StringBuilder x = new StringBuilder("a");
        x.append("b");
```

This may seem like overkill for one concatenation, but when this statement is run multiple times in a loop, it saves a lot of memory.