CS2030 Java Memory:

In a nutshell:

Java Memory works in this way:
Stack: Storage area for local variables (One per thread)
Heap: Where Classes and Objects are stored (One per JVM instance)
MetaSpace: Where static methods and variables are stored.

```
public class Memory {

	public static void main(String[] args) { // Line 1
		int i=1; // Line 2
		Object obj = new Object(); // Line 3
		Memory mem = new Memory(); // Line 4
		mem.foo(obj); // Line 5
	} // Line 9

	private void foo(Object param) { // Line 6
		String str = param.toString(); //// Line 7
		System.out.println(str);
	} // Line 8

}
```

How does memory work when a program is run?

1) All runtime classes are loaded into the heap space. When the main driver program is found, Java Runtime creates stack memory to be used by the main method thread.
2) We are creating primitive local variable at line 2, so it's created and stored in the stack memory of main() method.
3) A Object is created next which goes into the heap memory, with a local reference to that memory space created in the stack, Similarly for Memory()
4) Next when we call foo() a block in the top of the stack is created to be used by foo() and all the local variables involved.
5) A String is created in foo(), which adds to the heap space, and a reference is created in the foo() stack to that memory location.
6) Foo() terminates and memory allocated for foo() is free.
7) Then main terminates and the whole stack is terminated.
8) Java Runtime frees all the memory and ends the execution of the program.

Now in more detail:
Heap vs Stack:
Heap memory is used in all parts of the program, where as Stack memory seems to be localized to a single method call or any call dependent on the previous call.

Whenever a object is created, ie when the 'new' keyword is initialized, the Object is created in the Heap Space and all variables associated to the object stored within it, and a reference

to the object is stored in the heap where it is initialized. Stack memory only contains local primitives and references to objects inside heapspace.

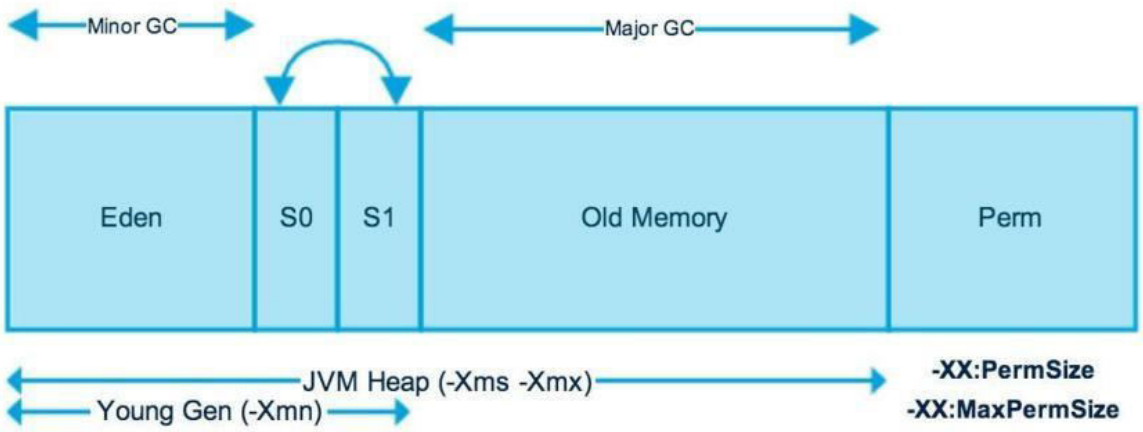
Objects in the heap are globally accessible while stacks are only accessible to the respective thread.

Stack Memory is temporary, while heap space lasts throughout the program.

When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.

Note that it is much more likely that the stack will run out of memory before the heap, and to circumvent StackOverflow errors, we can apply a technique called lazy evaluation in order to utilize the heap space for computational evaluations.

Java Heap in depth:



Java Heaps are further subdivided into portions of eden, s0/s1 , old memory and permgen space.

The eden space is where all objects are created, and they move to S0 and S1 space immediately afterwards.

As a program runs, some objects get dereferenced within the S0/S1 space.

After some time, JVM triggers a round of Minor Garbage Collection.

What is Garbage Collection?

Java employs a graph searching algorithm to perform garbage collection, a task which traverses the memory references of objects created within the heap, and flags any objects that do not have references for Garbage Collection, which deletes dereferenced objects. This process pauses the program momentarily.

As the program continues to run, multiple rounds of garbage collection occur, and some objects continue to survive in the heap space. These objects are then promoted and moved from S0/S1 space into the Old Memory heap space.

As Garbage Collection cycles are relatively computationally expensive, the Old Memory Heap Space is not usually affected by cycles of minor garbage collection. However, once enough cycles of Minor GC occur, the whole memory is traversed in a round of Major Garbage Collection, which causes a relatively large pause in the program.

After the major GC concludes, the program can then continue computations until program termination.

On PermGen/MetaSpace:
With Java 8, there is no Perm Gen, that means there is no more "java.lang.OutOfMemoryError: PermGen" space problems. Unlike Perm Gen which resides in the Java heap, Metaspace is not part of the heap. Most allocations of the class metadata are now allocated out of native memory. Metaspace by default auto increases its size (up to what the underlying OS provides), while Perm Gen always has fixed maximum size. Two new flags can be used to set the size of the metaspace, they are: "-XX:MetaspaceSize" and "-XX:MaxMetaspaceSize". The theme behind the Metaspace is that the lifetime of classes and their metadata matches the lifetime of the classloaders. That is, as long as the classloader is alive, the metadata remains alive in the Metaspace and can't be freed.

Hence this is how Java handles the back end of memory allocation, which made it so popular with programmers at the time whom had to do as such manually.