JAVA MEMORY MANAGEMENT

* There are 2 types of memory which Java creates. JVM manages these both :-
  1) Stack
  2) Heap

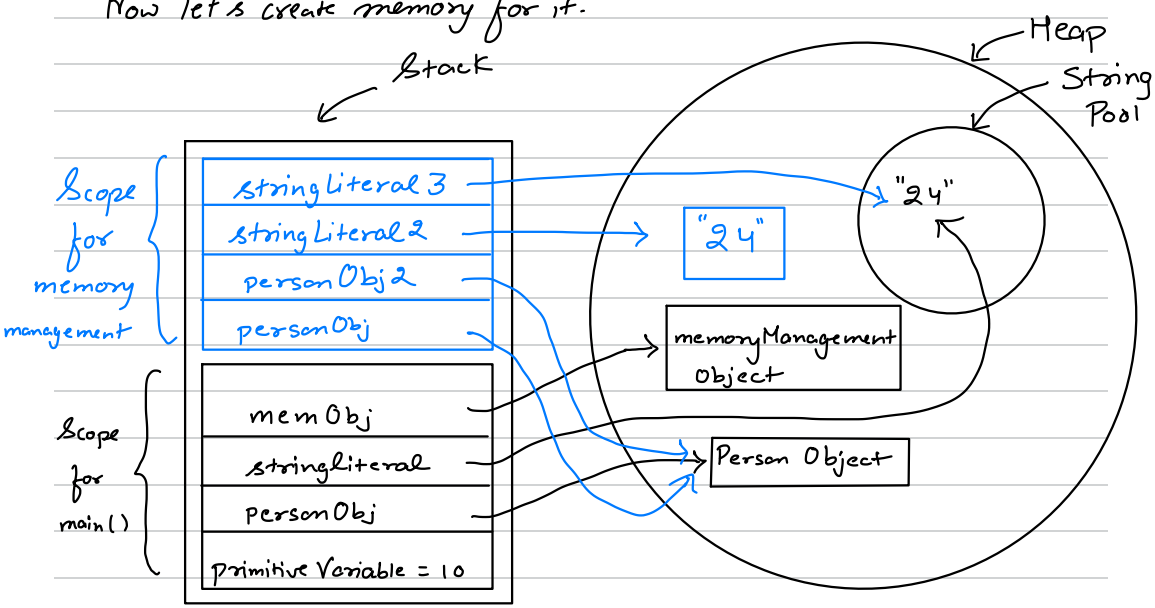
=> Stack Memory
- Stack stores temporary variables & separate memory block for methods.
- Store primitive data types.
- Store reference of heap objects
  - Strong reference
  - Weak reference
    - Soft reference
- Each thread has its own stack memory
- Variables within a SCOPE is only visible and as soon as any variable goes out of the SCOPE, it gets deleted from the stack (in LIFO order)
- When stack memory goes full, it throws "java.lang.StackOverflowError"

=> Heap Memory
- Store objects & There is no order of allocating the memory
- Garbage Collector is used to delete unreferenced objects from the heap.
  - Mark & Sweep Algorithm
  - Types of Garbage Collector i.e Single GC, Parallel GC, G1, and CMS (concurrent mark & sweep)
- Heap memory is shared with all the threads.

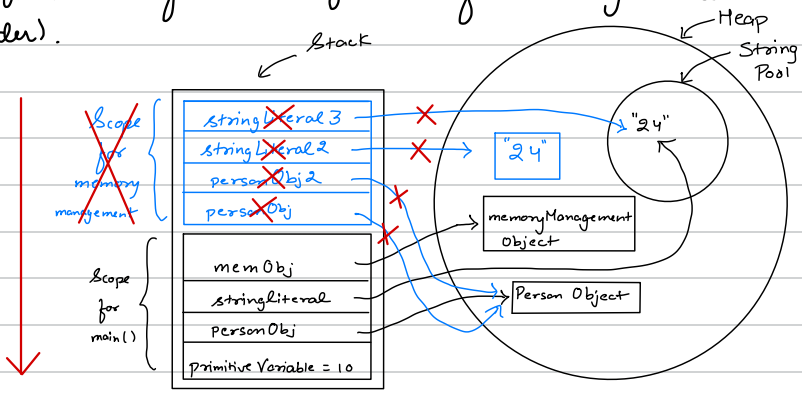
* Now let's understand both with the example

```java
public class MemoryManagement {

    public static void main(String args[]) {

        int primitiveVariable1 = 10; // primitive data type
        Person personObj = new Person(); // object
        String stringLiteral = "24"; // string literal
        MemoryManagement memObj = new MemoryManagement();
        memObj.memoryManagementTest(personObj);

    }

    private void memoryManagementTest(Person personObj){

        Person personObj2 = personObj; // reference to object
        String stringLiteral2 = "24"; // string literal
        String stringLiteral3 = new String( original: "24");
    }                                  ↳ String Object
}
```

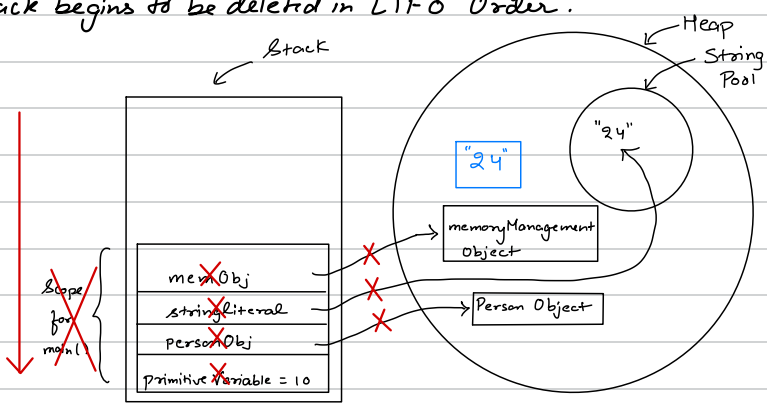
So, we have a simple class with various types of data is stored.
Now let's create memory for it.



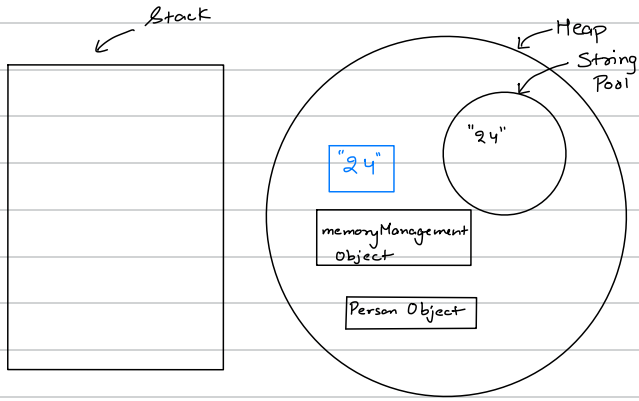
Now as soon as we encounter the closing bracket of memoryManagementTest method, its scope ends, it will delete its scope so all of the blue portion of stack gets deleted (in LIFO Order).



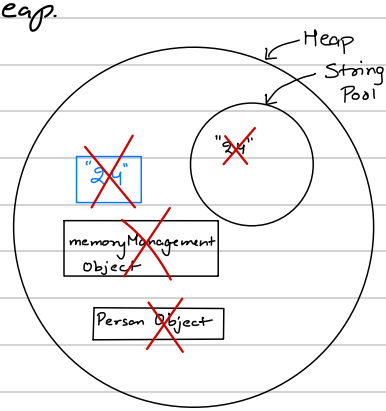
Now control comes back to main() method. Since nothing is there after calling memoryManagementTest api, we encounter the closing bracket which means the scope of main ends & it's portion in stack begins to be deleted in LIFO Order.



So now the stack is cleared & all the references are deleted from the stack as well. Now the memory looks like this



Now the stack is cleared & all the references are deleted but the objects are in the heap. So that's where garbage collector's work comes. Garbage Collector will delete all the unreferenced objects from the heap.



Garbage Collector runs periodically & JVM controls when to run the garbage collector. We can also tell the JVM to run the garbage collector using System.gc() but this doesn't guarantee that GC will run & that is why all of this is kla automatic memory management. The frequency of GC running is directly proportional to how much of the heap memory is currently full.

⇒ Type Of References :-

* Strong Reference :-

- It is when a variable in stack is referencing an object in Heap memory.
- Till the time the reference exists, GC won't be able to delete the object from the Heap Memory.

For eg:-

```
Person pobj = new Person();
```

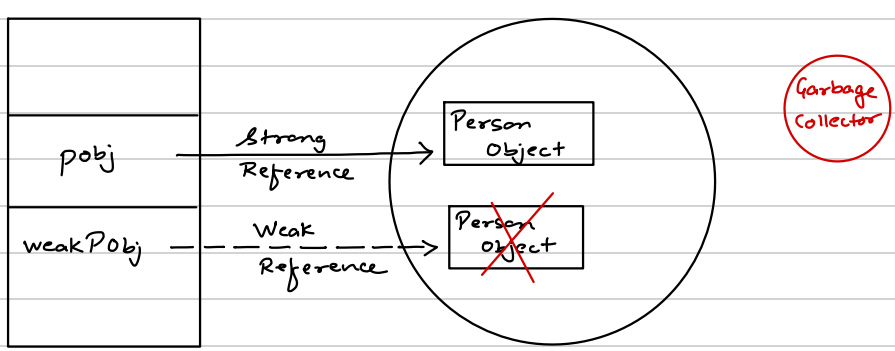
So here pobj has strong reference to a Person Object in the heap memory & till the time this reference exists, GC won't be able to delete Person object from Heap.

* Weak Reference :-

- In weak reference also the reference exists to an object in the heap but as soon as GC runs the object is deleted from heap memory even if some variable is referencing this object from the stack. The variable in the stack will get null if it tries to access the object post GC run.

For eg:-

```
WeakReference <Person> weakPobj = new WeakReference <Person> (new Person);
```



* Soft Reference :-

- It is a type of weak reference but the difference is that in this case the object will be deleted only when there is shortage of space in Heap. So GC is allowed to delete a soft reference but it'll keep the object if sufficient space is there in heap.

* Reference can be changed by referencing a current object to a new variable.

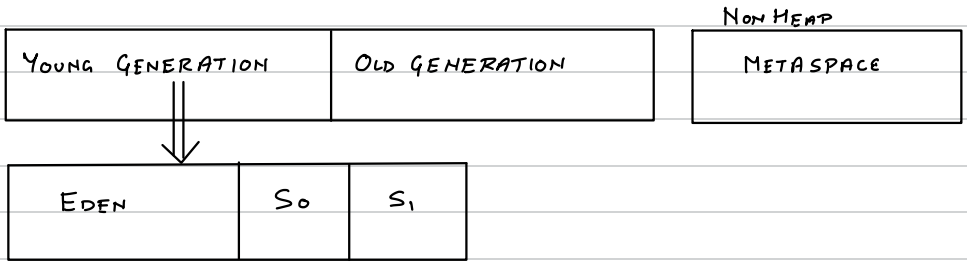
Eg:- Person obj1 = new Person (); // created new object
Person obj2 = new Person (); // created another object
obj1 = obj2;

Now obj1 will have a reference to the object of obj2 in heap & when GC runs, the earlier object which obj1 was refering to will be deleted.

⇒ Heap Memory

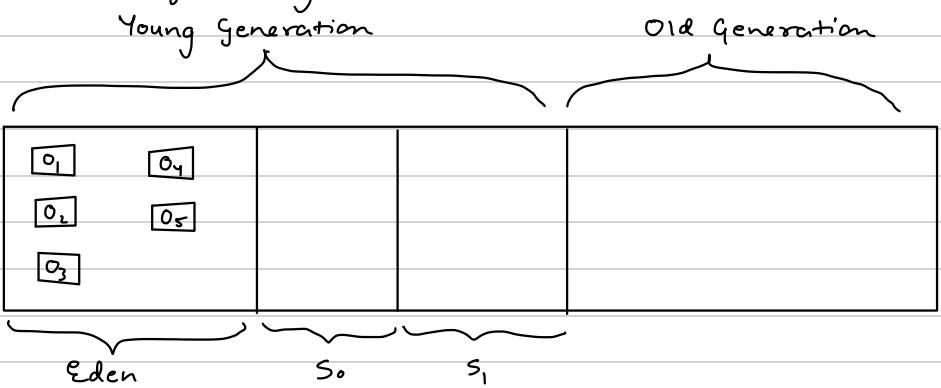
Heap memory is divided into two parts i.e Young Generation and Old Generation. There is also one more part which is generally known as Non Heap (Metaspace). Before Java 7 it is called permgen but now it is not used & it is used as metaspace.

Now young generation is further divided into 3 parts named Eden, S0 (Szero) and S_1. Szero & S_1 are known as survivor space



Now let's see when we create an object, what happens to it.

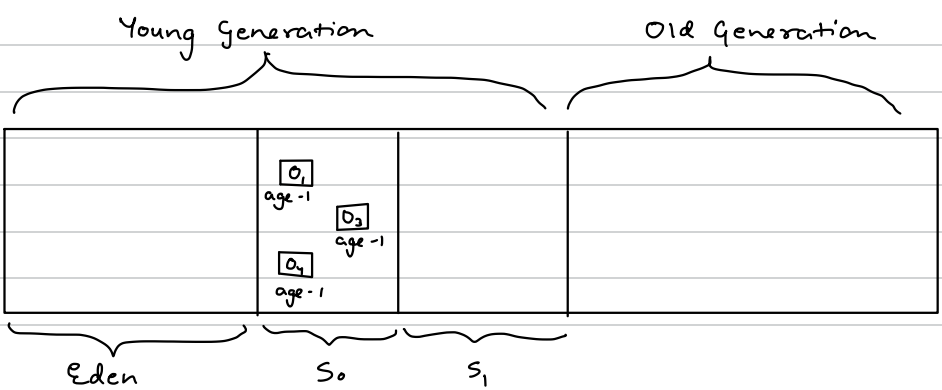
* Whenever a new object is created, it goes to Eden first. Let's say we've created 5 objects (O_1, O_2, O_3, O_4 & O_5). They'll be created in Eden first. So now heap memory looks like.



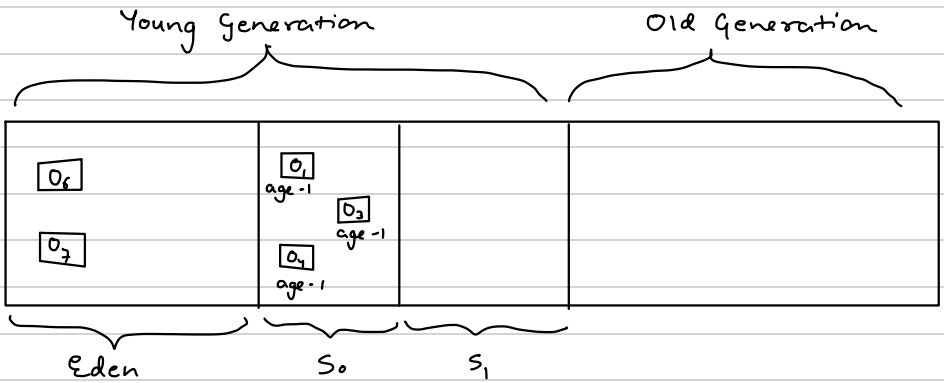
So 5 objects are created inside Eden.

Now let's say Garbage Collector run & there is no reference to O_2 & O_5 in the Heap Space. So now GC will use Mark & Sweep Algorithm i.e in Mark, GC will mark the objects which no more have reference & then Sweep in which it'll do 2 things:

- First remove de referenced objects (O_2 & O_5) from the memory
- Move the rest of survivor objects into one of the survivor space ie S_0 or S_1 and adds age to the objects. So after GC runs, heap now looks like:



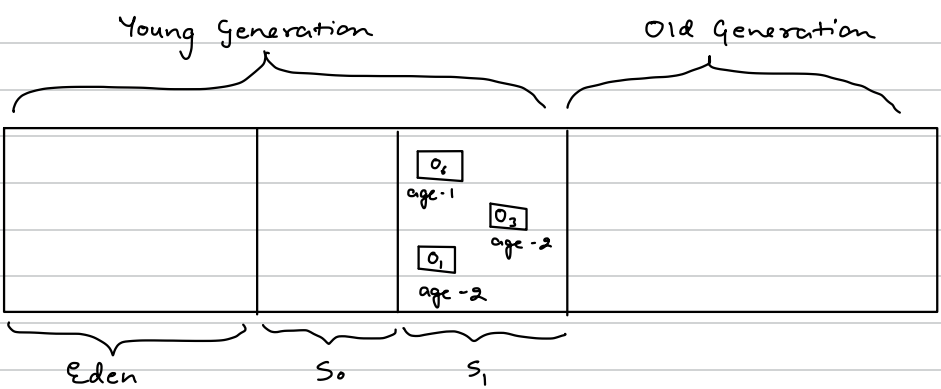
Now GC has ran once. This whole process is called minor GC as it happens very periodically & very fast.
Let's now create 2 more objects O_6 & O_7. So heap now looks like



O_6 & O_7 are now created in Eden. Now let's say the GC ran again & this time no reference is there for O_4 & O_7. So now GC will now:-

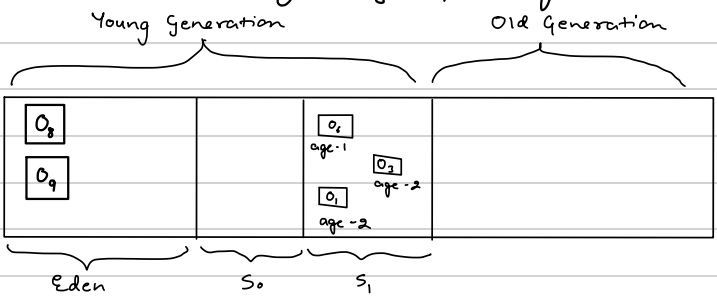
- Mark O_4 & O_7
- Deletes O_4 & O_7
- Moves O_1, O_6 & O_7 (survivors) to S_1 with corresponding ages.

Therefor post this minor GC, the heap looks like:



So at one time Eden would be completely free after the GC & one of the survivor space (S_0 or S_1) would be free and we put data alternatively in S_0 & S_1, along with the respective age.

Now let's create two more objects O_8 & O_9 & heap looks like:-



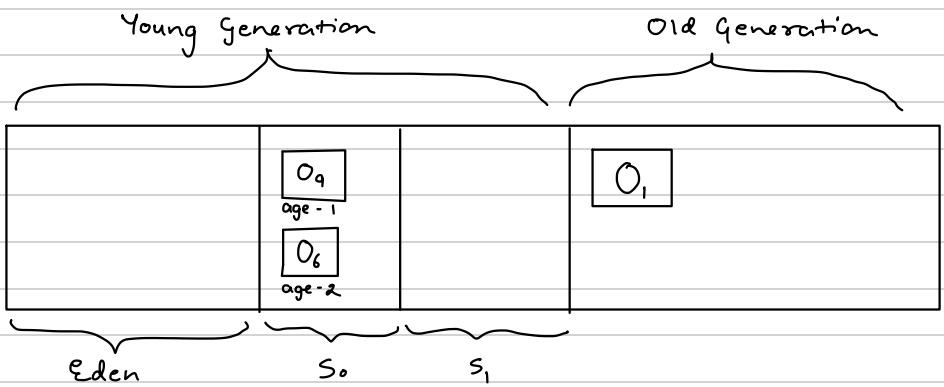
Now let's say I've set the threshold age to 3

Therefore now objects with age 3 needs to be promoted. Promotion means that now the objects with age 3 will be moved to Old Generation

Let's now run GC assuming that there are no more references to O_3 & O_8 hence will be deleted. So GC will now

- Mark O_3 & O_8
- Delete O_3 & O_8
- Move the survivor objects (O_6 & O_9) to S_0 & since O_1 still have reference with age 3, it'll be promoted i.e moved to old generation

Now the heap looks like:



In the old generation only difference is that here it is called major GC because the GC in Old Generation won't run too periodically. So in Old Generation the GC runs less periodically as compared to young gen and the objects in old gen are kind of big objects that are used too frequently & these might have a lot of references pointing to them.

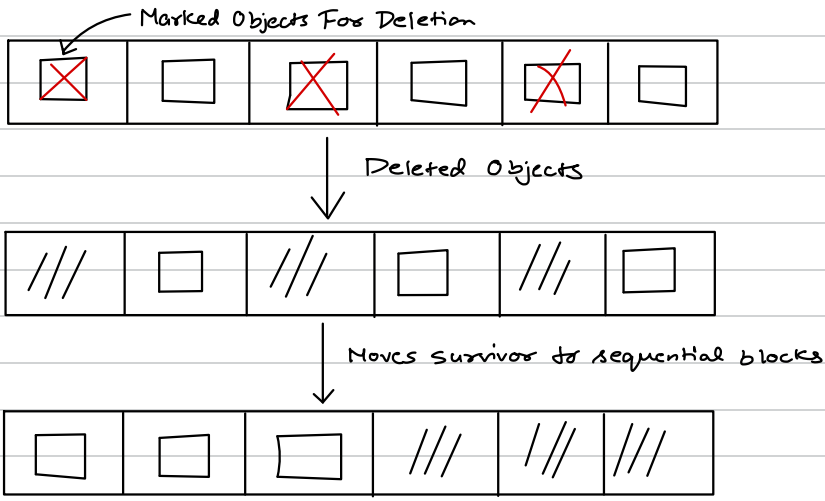
Now let's come to metaspace. What will it store?
The metaspace contains :-

- Class variables
- Class metadata (basically stores info about class from which objects can be created)
- Constants

* Prior to Java 8, there is a fixed space kla perma gen which used to store the data that is stored in meta space & since it was non expandable, once it filled, we'll get out of memory error.
  But now from Java 8, metaspace is there which is different from heap & is expandable.

⇒ Garbage Collector Algorithms
* Mark & Sweep Algorithm (Already Discussed)
* Mark & Sweep with compact memory
  In this once the GC runs, the remaining objects are put in a sequential memory block leaving another sequential block free to put objects

  Let's take this example



So this is kla mark & sweep with compaction

⇒ Versions Of GC

1) Serial GC
   - In this only one GC thread is working for both minor and major GC.
   - It's disadvantage is that due to the single thread, GC will be slower but since GC is expensive (because when GC work starts, all application threads pauses). So this is one of the disadvantages of serial.

2) Parallel GC
   - In this there are parallel threads running based on the CPU configuration. So GC works faster. So the application threads will be in paused state for lesser time.

3) Concurrent Mark & Sweep
- In this the GC will try its best to run concurrently with application threads but there is no guarantee. In this there is no memory compaction happens.

4) G1 Garbage Collection
- This is a better version of concurrent mark & sweep in which GC will try not to stop / pause the application thread and supports memory compaction as well.

* Currently Java 8 is using parallel GC
* In latest Java Versions using CMS & G1, there is a high chance that pause time is minimal thus increasing throughput & decreasing latency.