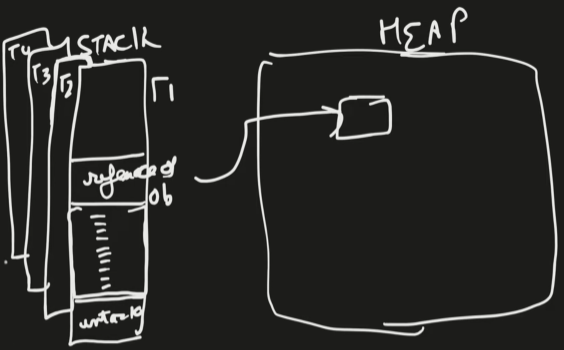
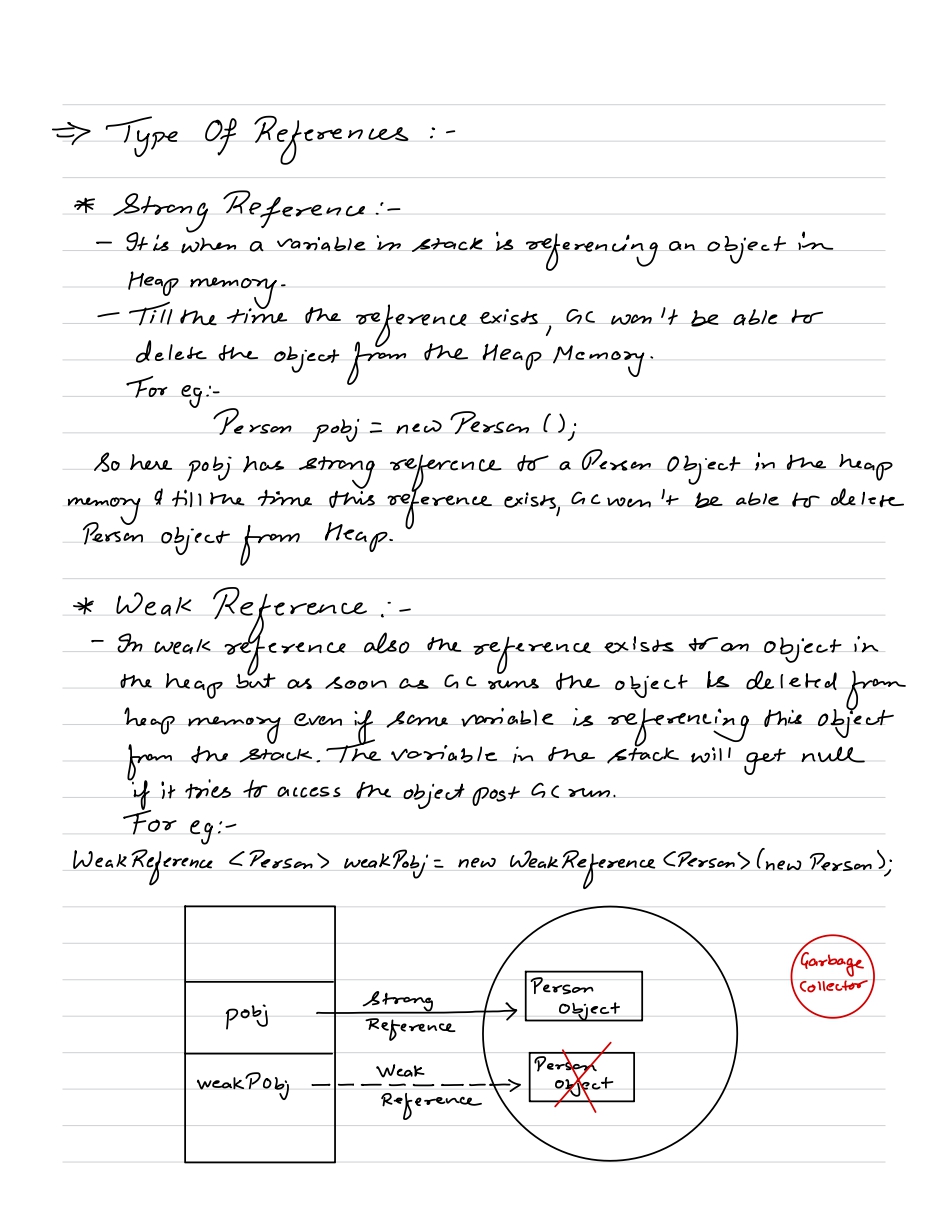
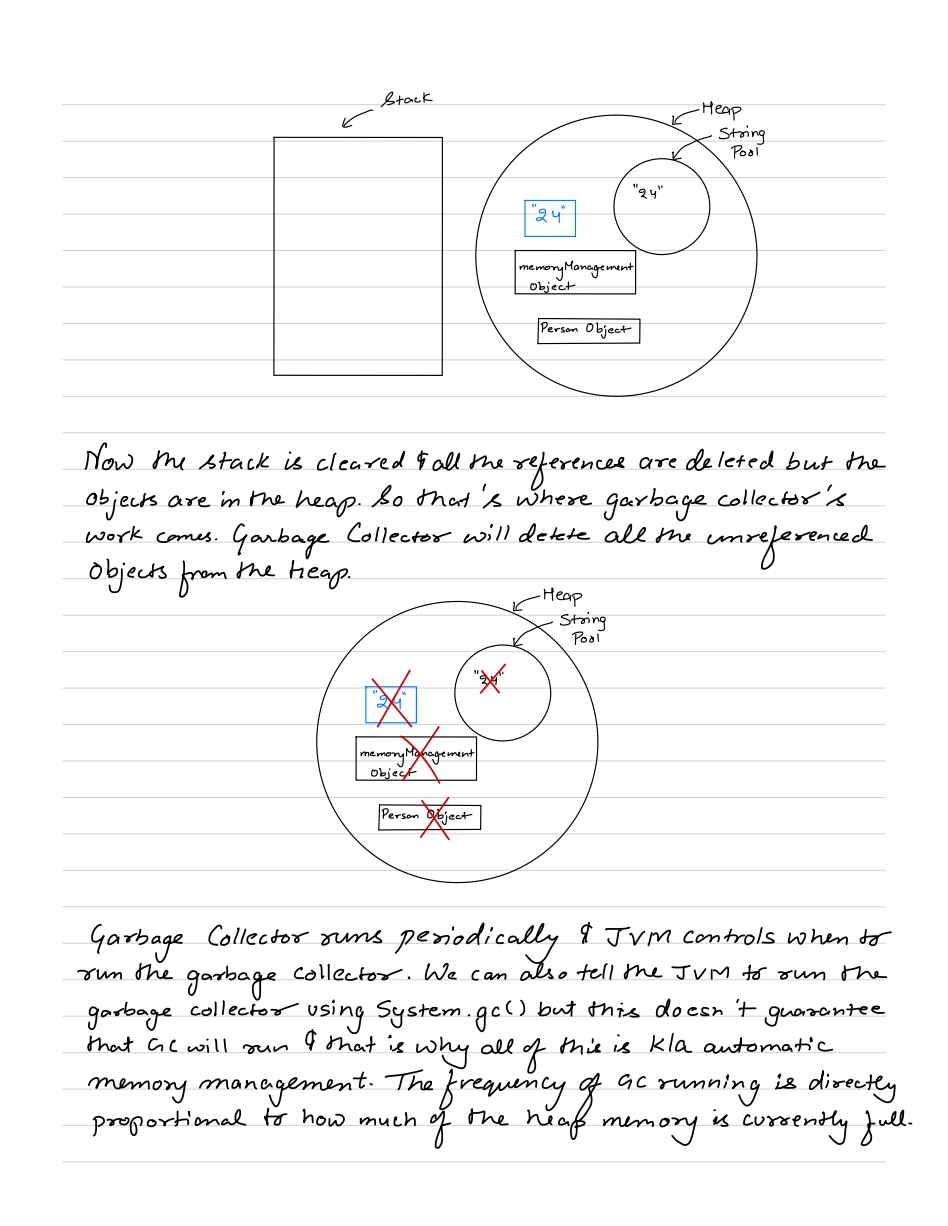
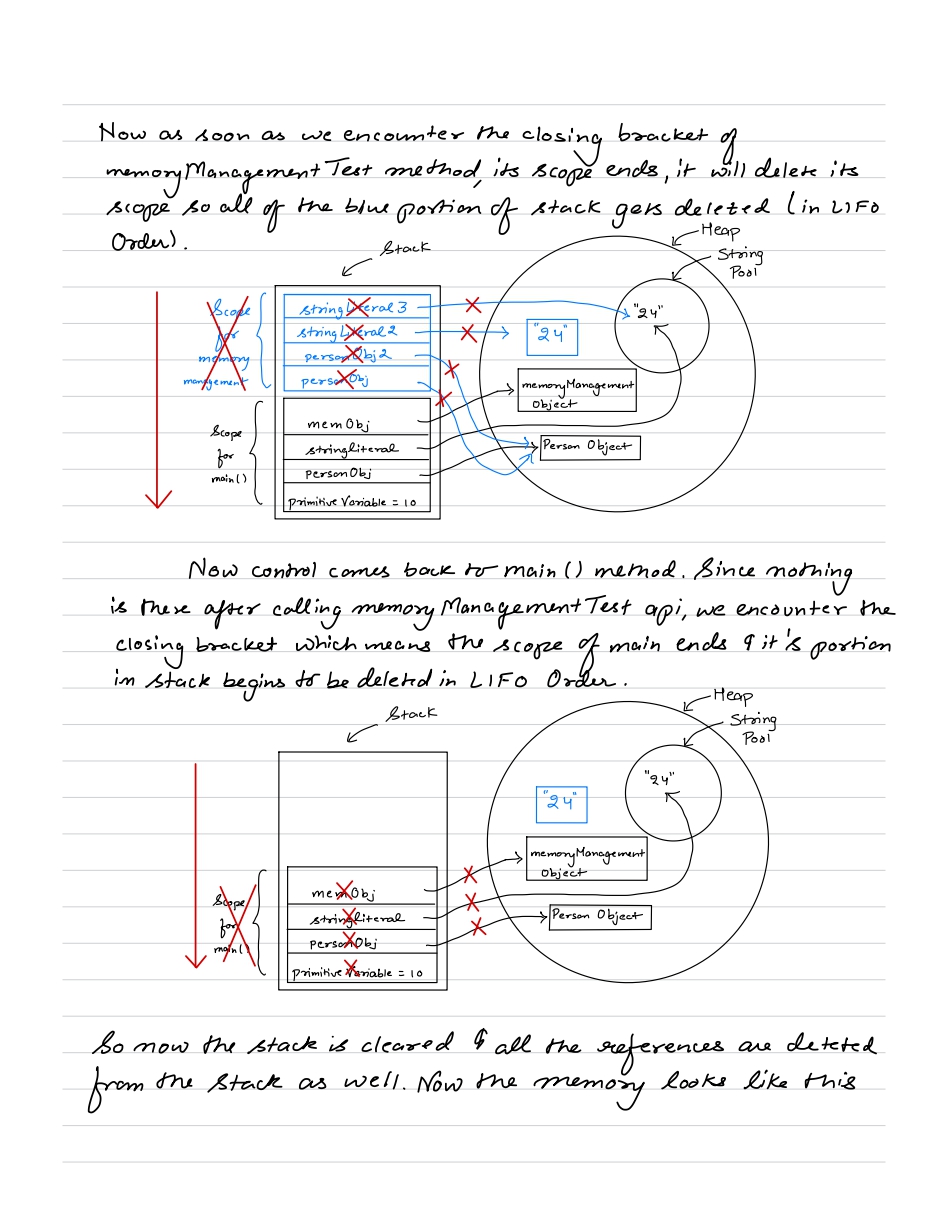
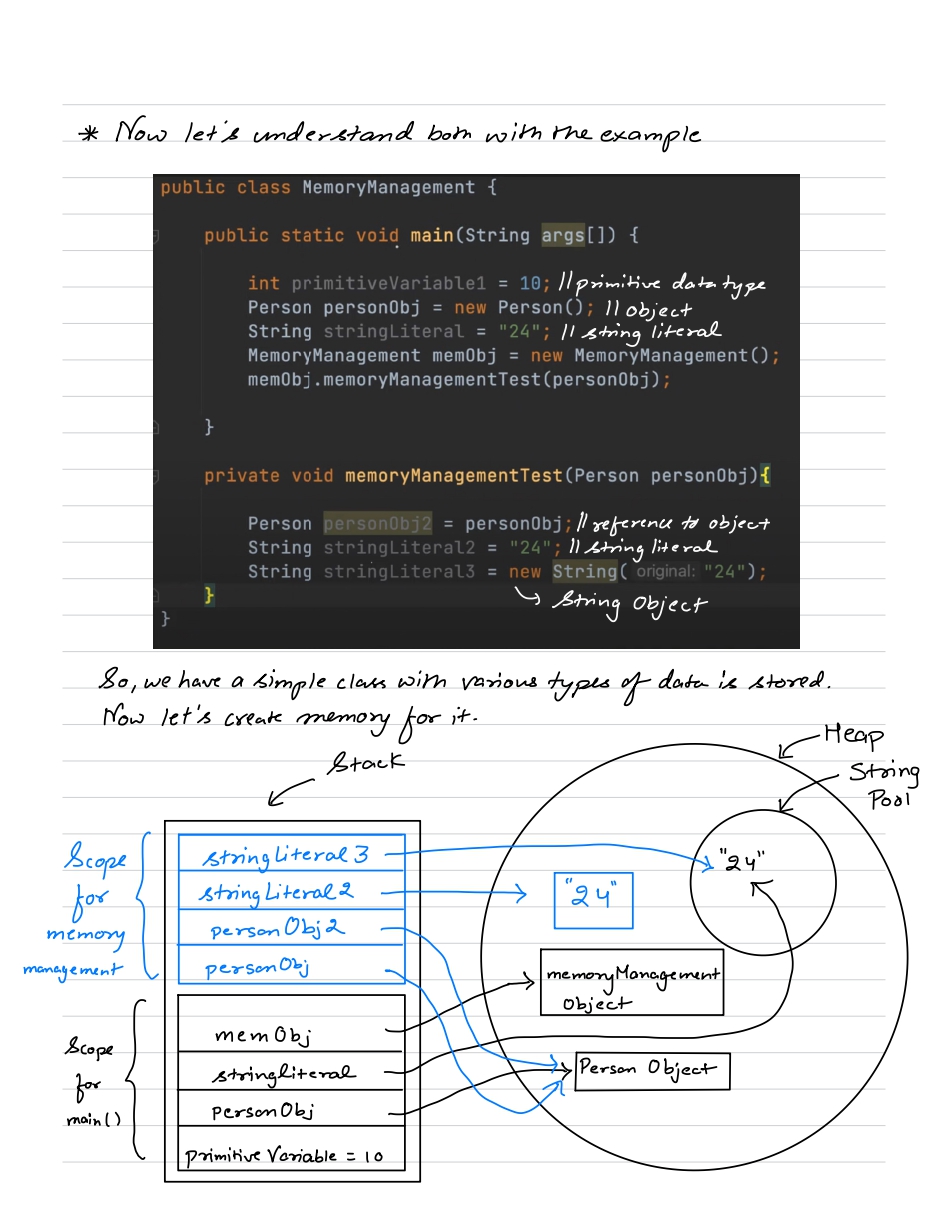
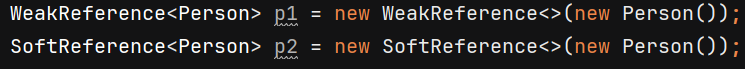
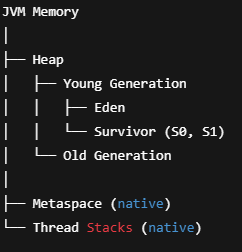
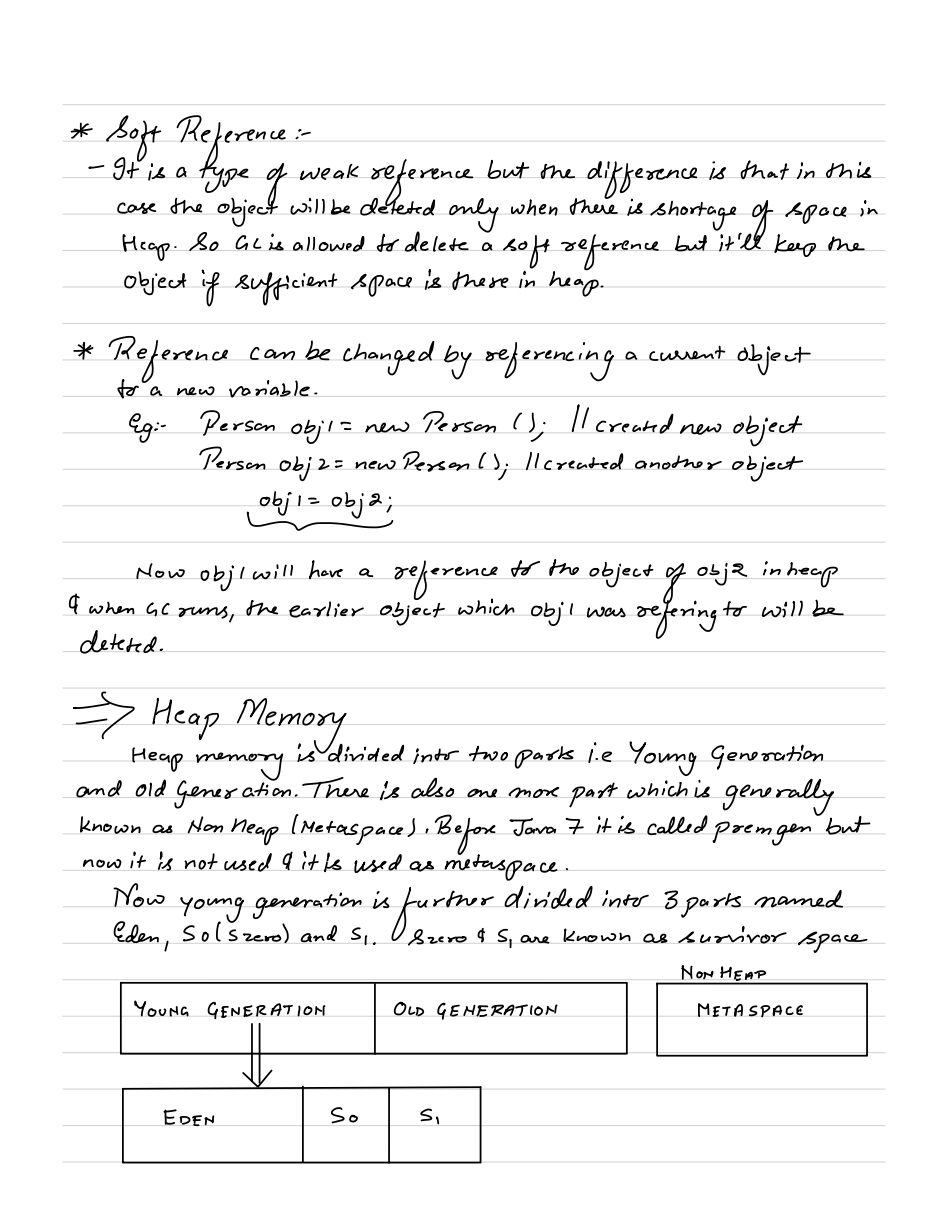
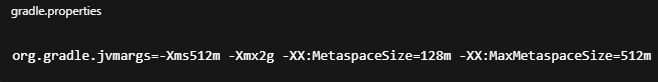


By memory we mean RAM

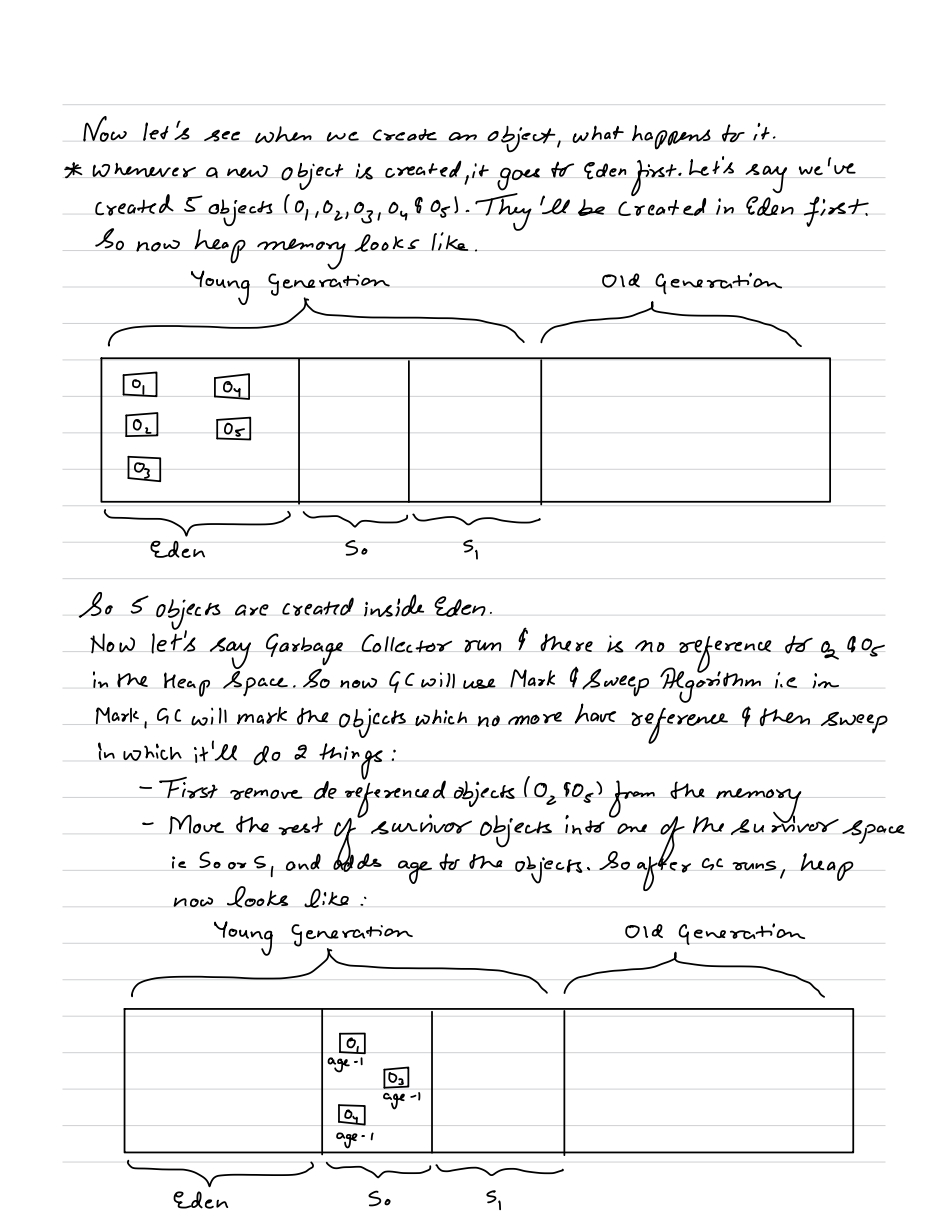
For each method, stack memory stores a memory frame for it and all variable inside that method are stored in that memory frame.



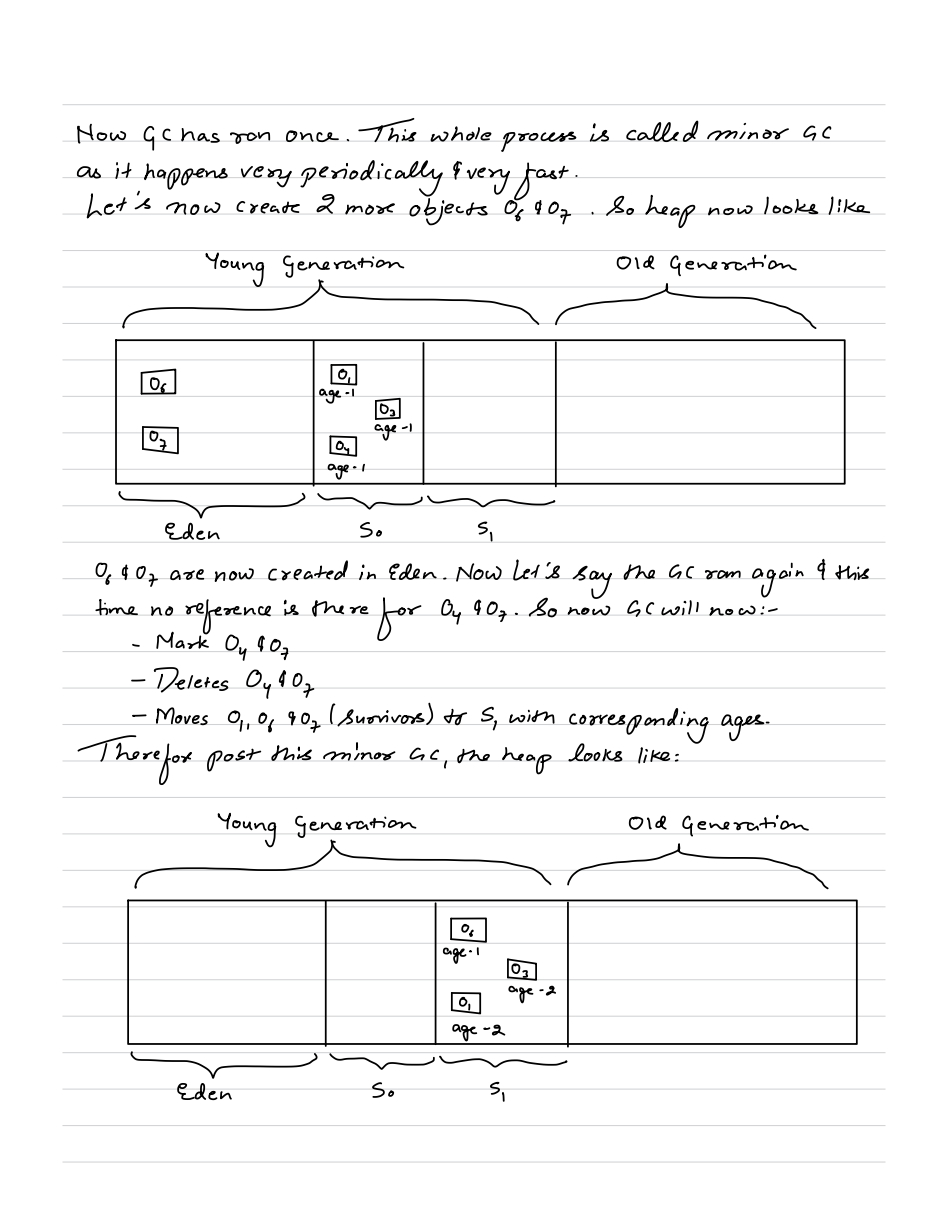
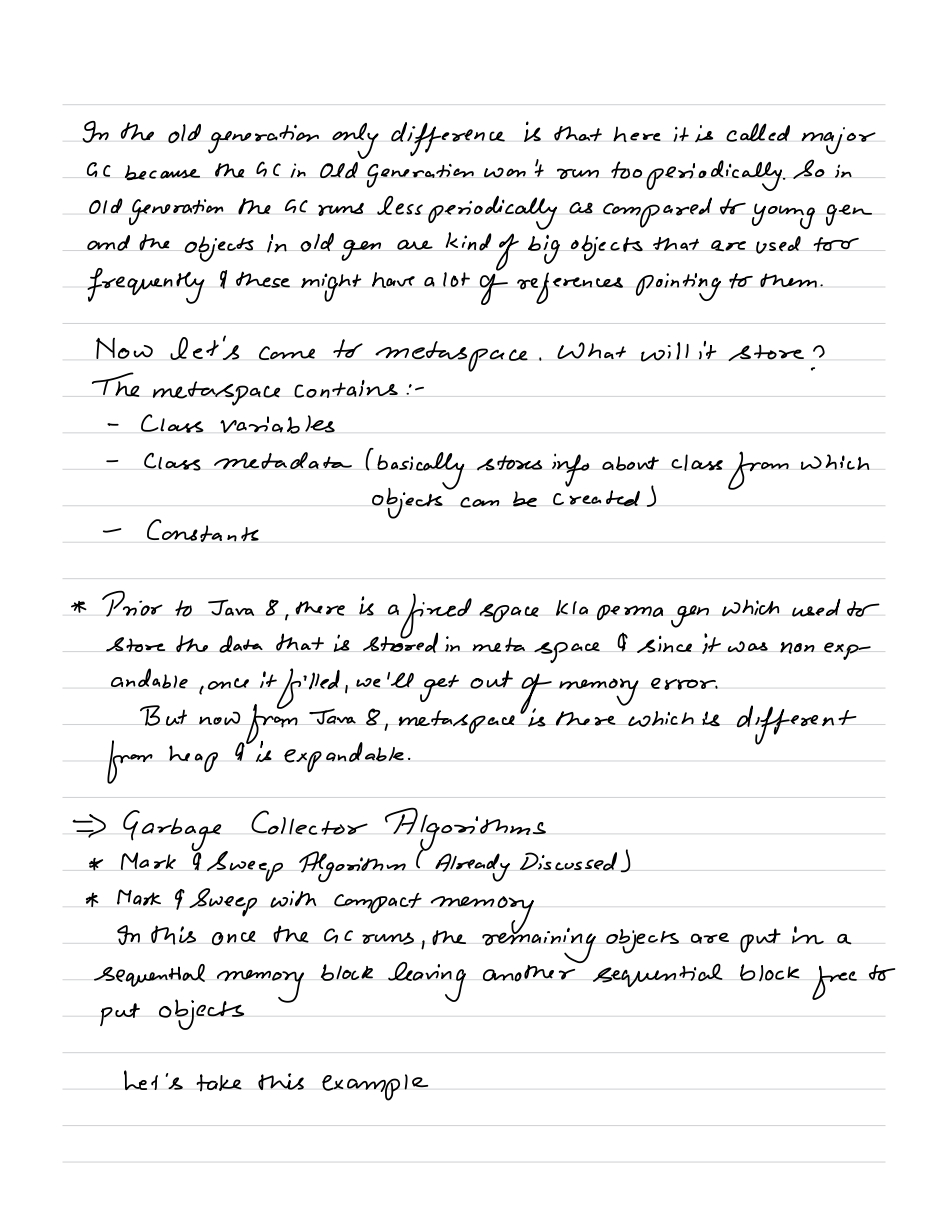
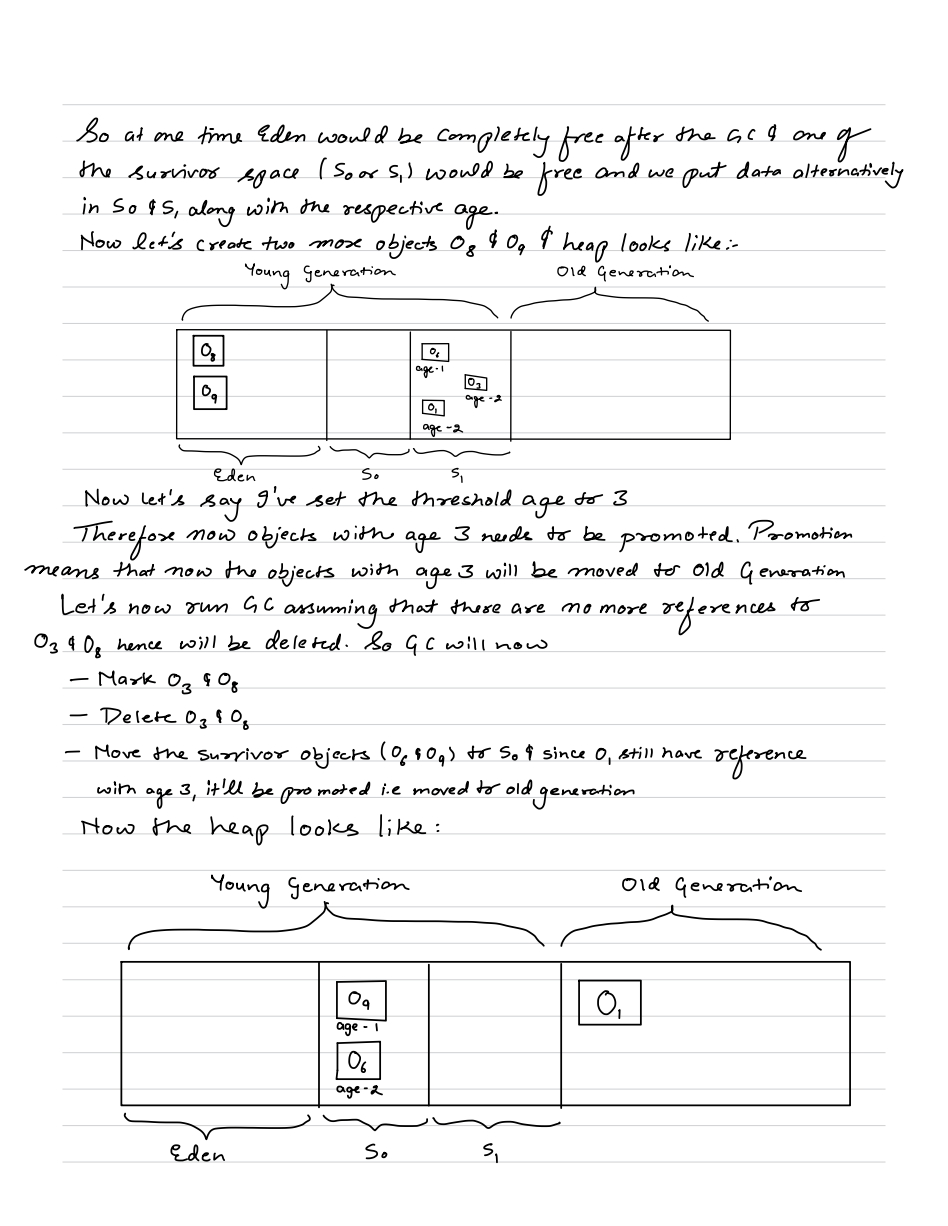
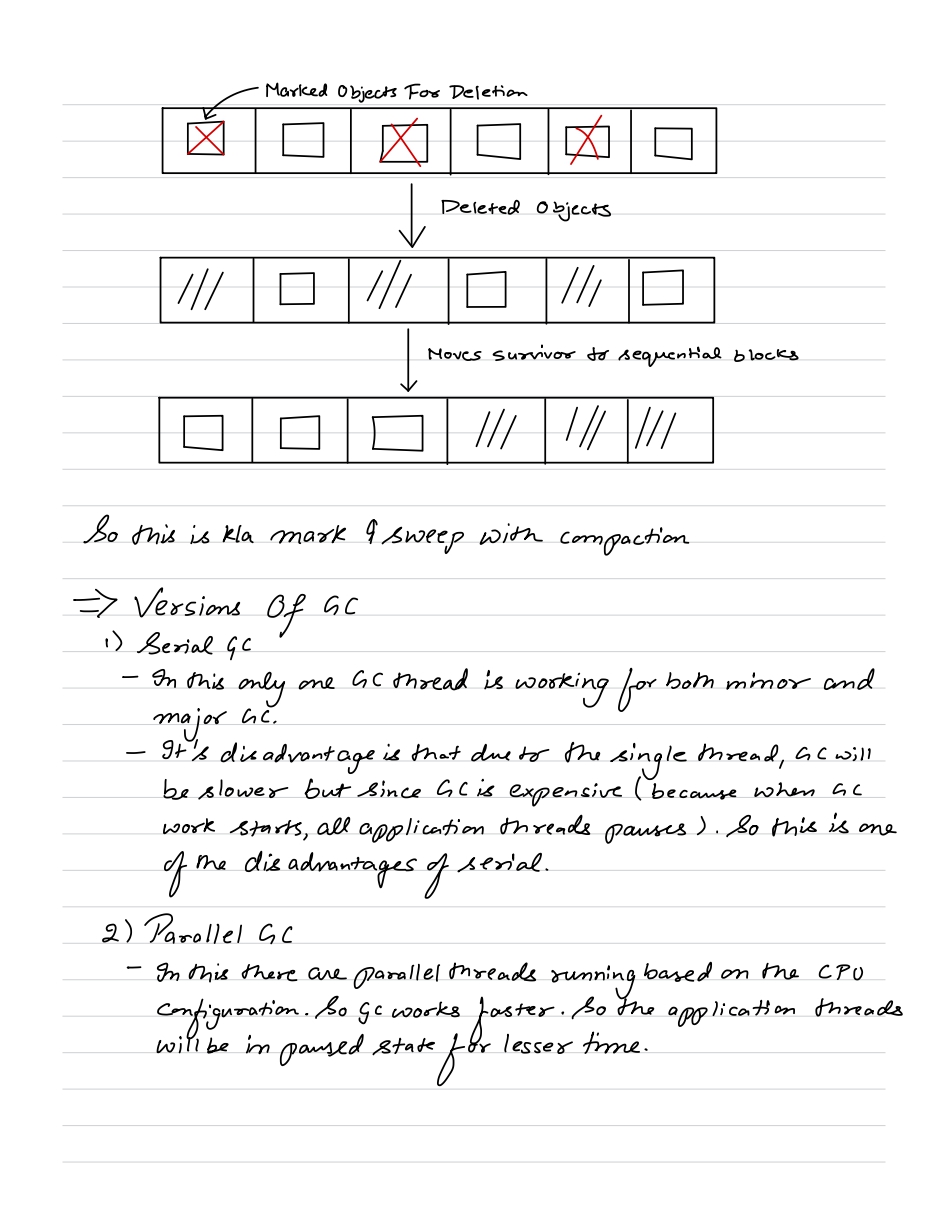


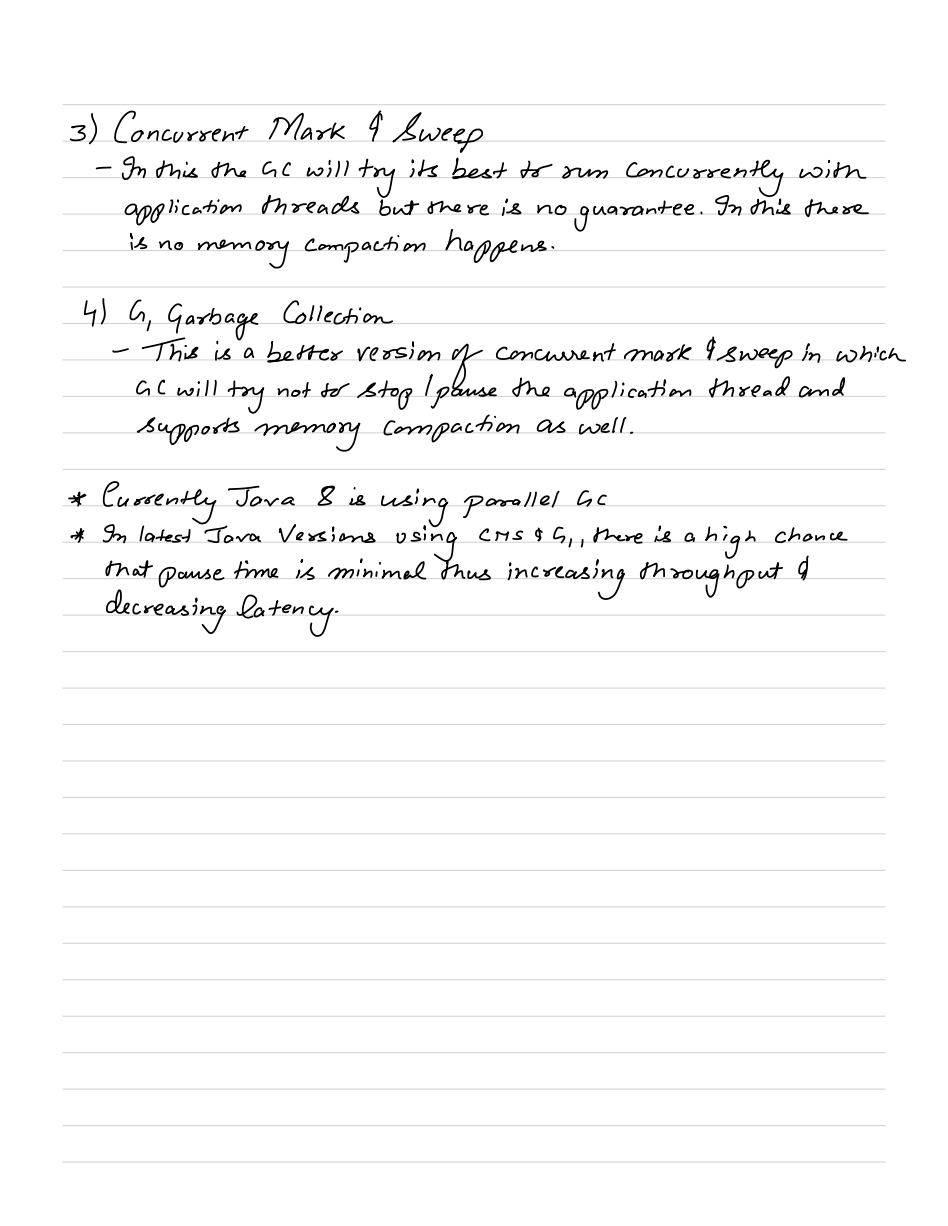
 

PremGen (before Java 8) was considered a part of Heap memory.

Young Generation uses Minor GC to clean short-lived objects. It happens very fast and runs more frequently. Pause time is less.

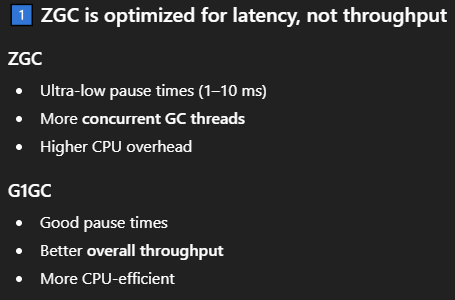
Old Generation uses Major GC to clean long-lived objects. It is time taking and runs less frequently.

During GC, JVM freeze application threads to get a consistent view of object references. This happens in both Major and Minor GC. But for Major GC pause time is longer.  
  
  
  
  
As soon as JVM needs certain class, it will load in metaspace and remove it when it doesn't need.  
  
  
Parallel GC is default in Java 8.   
  
In CMS, concurrent GC threads run alongside application threads to minimize pause time by Major GC. But it does not gaurantee that application threads will not stop.

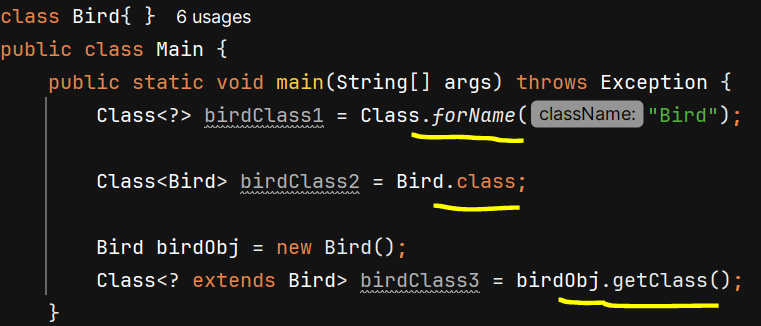
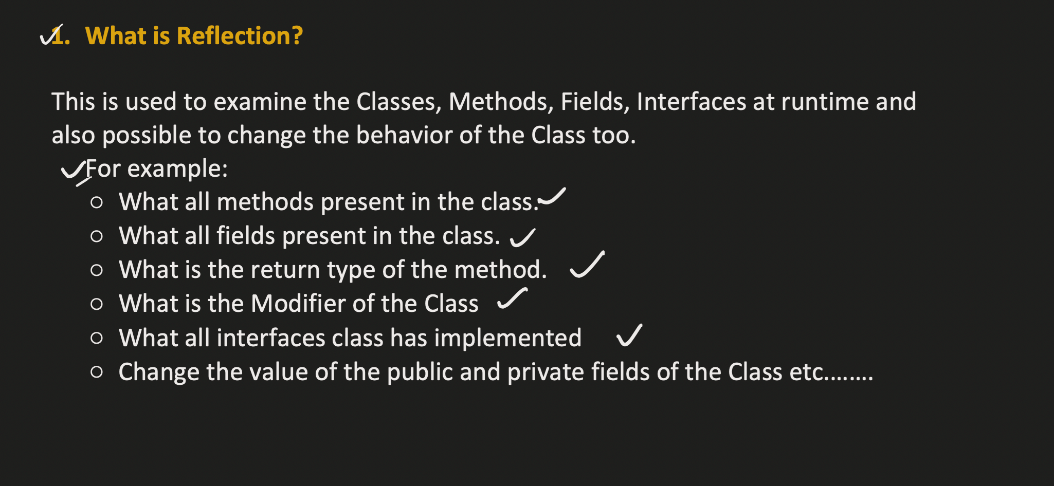


G1GC is default in java 17/21/22/23.

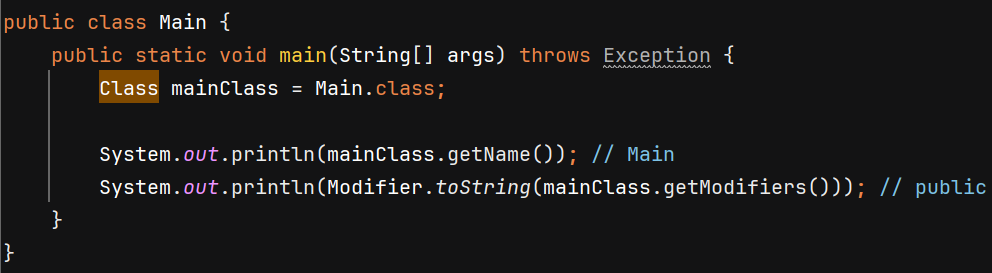
**ZGC** is a low-latency garbage collector that performs concurrent marking and compaction to keep GC pauses under 10 ms, even for very large heaps.



We configure which GC to use in **gradle.properties** file like –

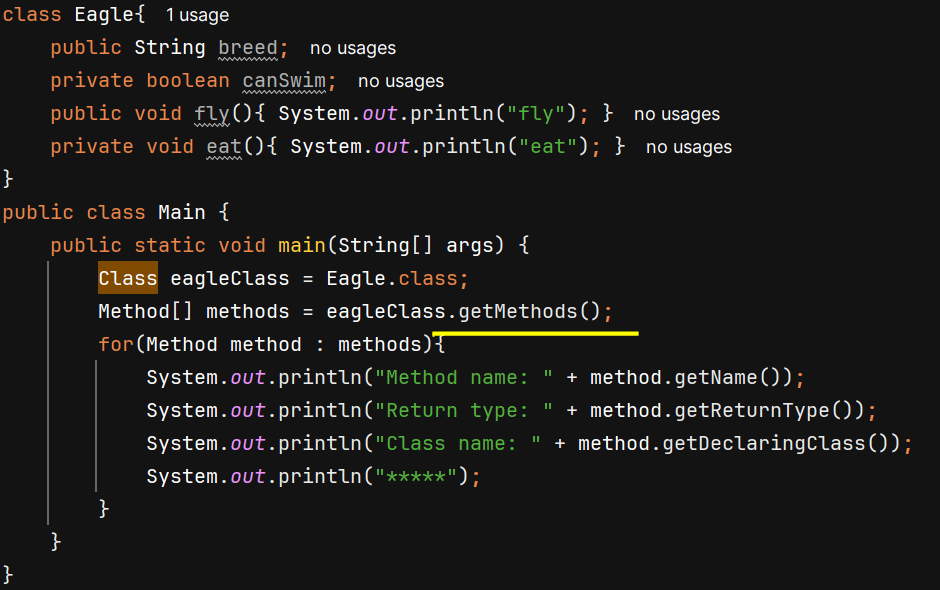


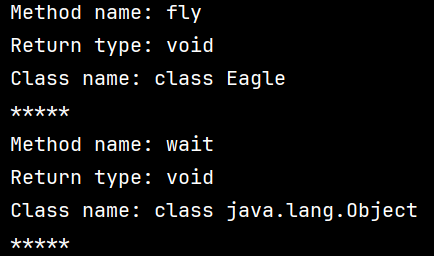
**Reflection of Classes**



mainClass.getModifiers() return public because class is public.

**Reflection of Methods**

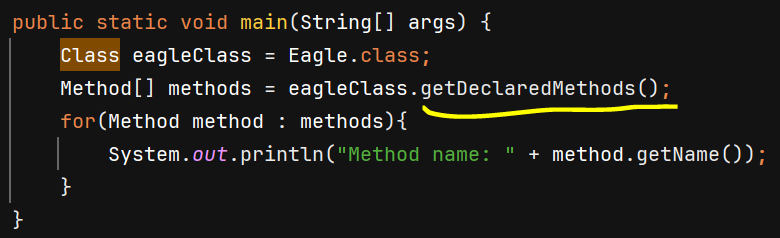
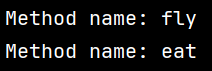




Every class is internally child class of Object.

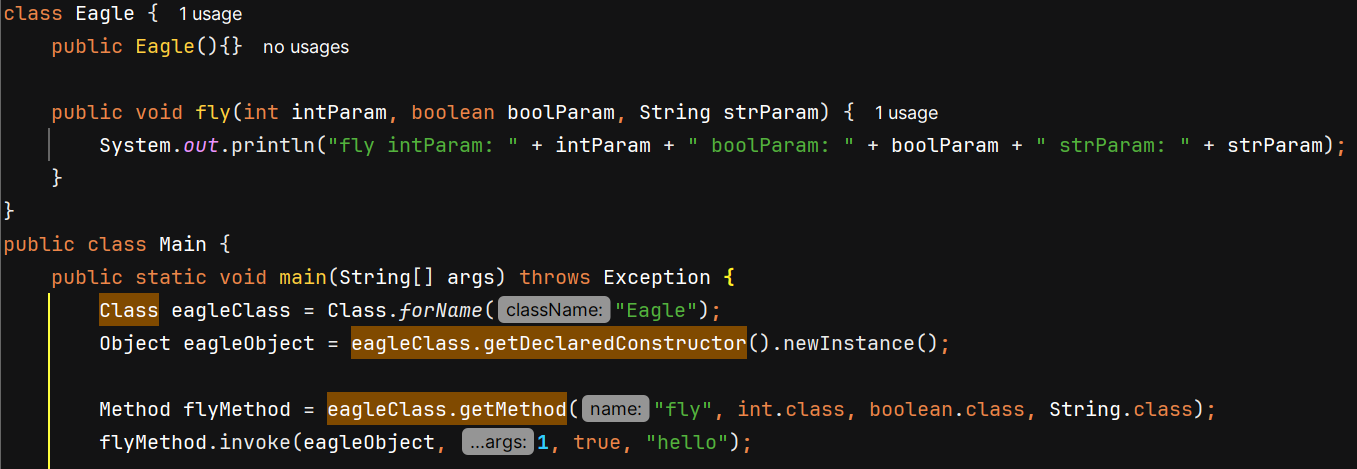
getMethods() will return all the public methods in that class and also in its super class.

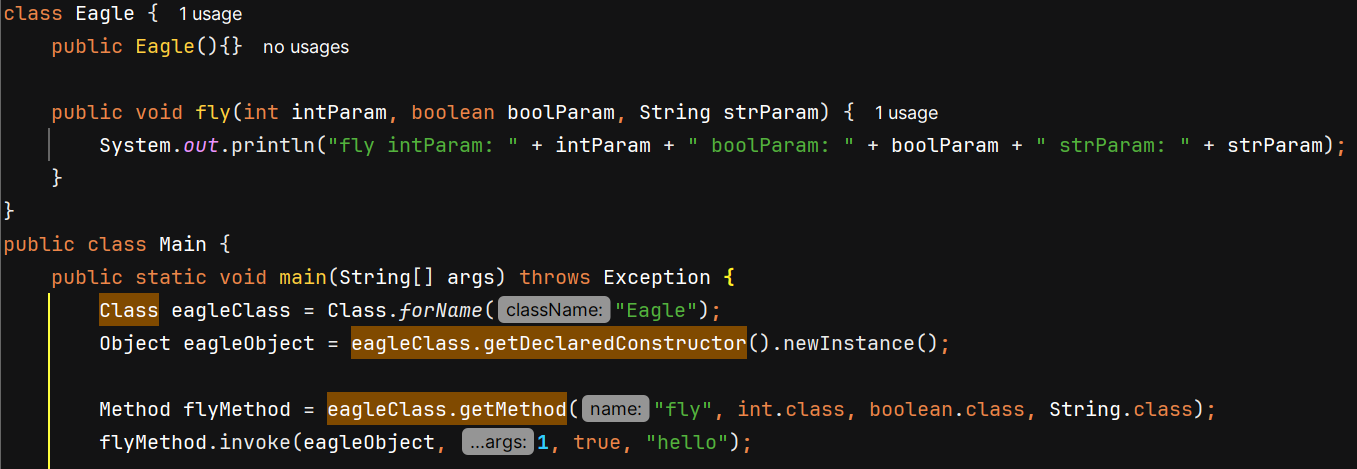
getDeclaredMethods() return all public and private methods within Eagle class only

 O/P - 

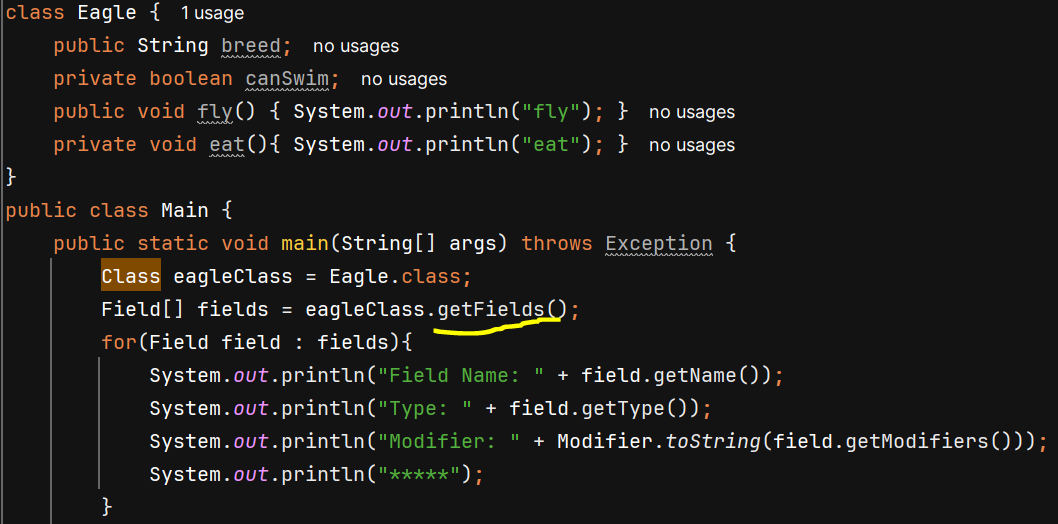
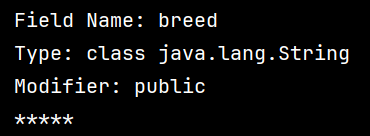
**Invoking Method using Reflection**

To invoke a method, we need object so we get it using **newInstance()** which internally invoke constructor of that class and return its object.



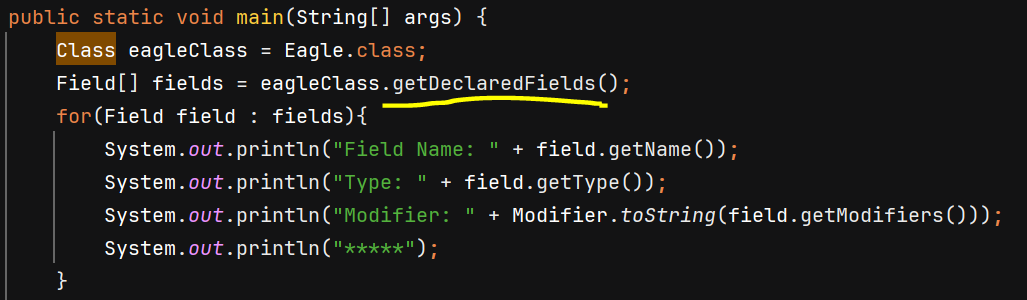
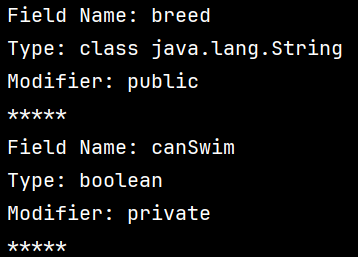


**Reflection of Fields**

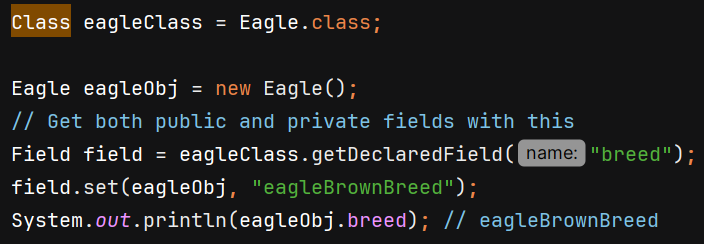
O/P – 

getFields() will only return public fields of class

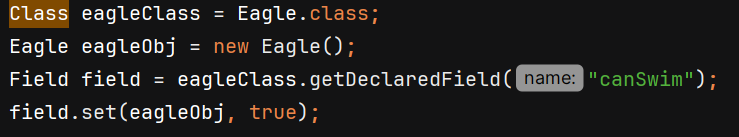
getDeclaredFields() will return both public and private fields

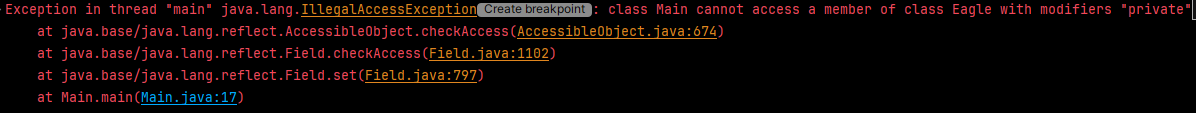
O/P – 

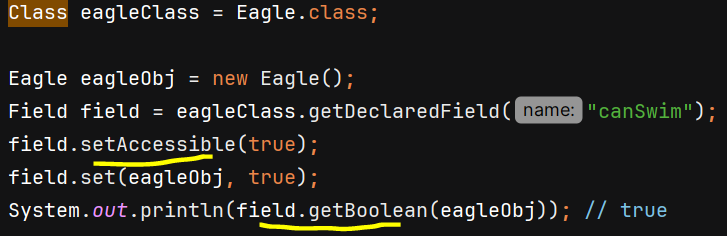
**Setting value of Public field**



**Setting value of Private field**

Incorrect way – 

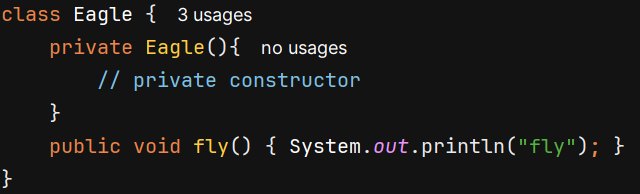
We get exception –

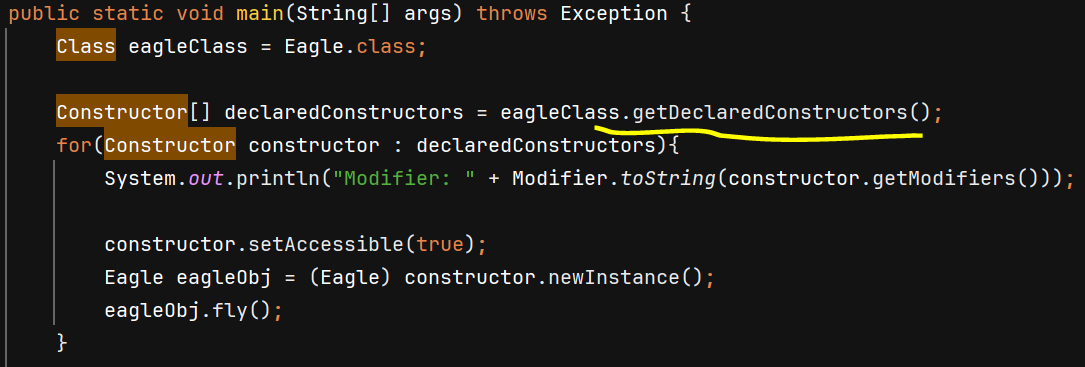
Correct way – 

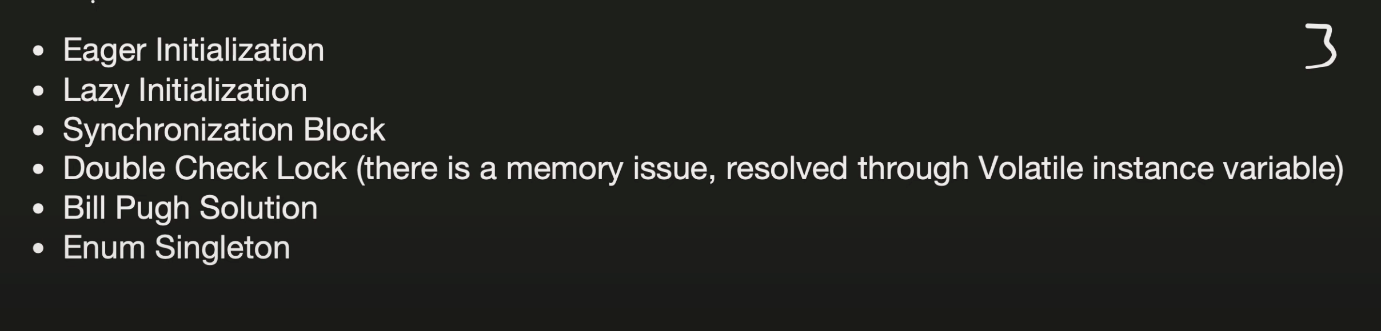
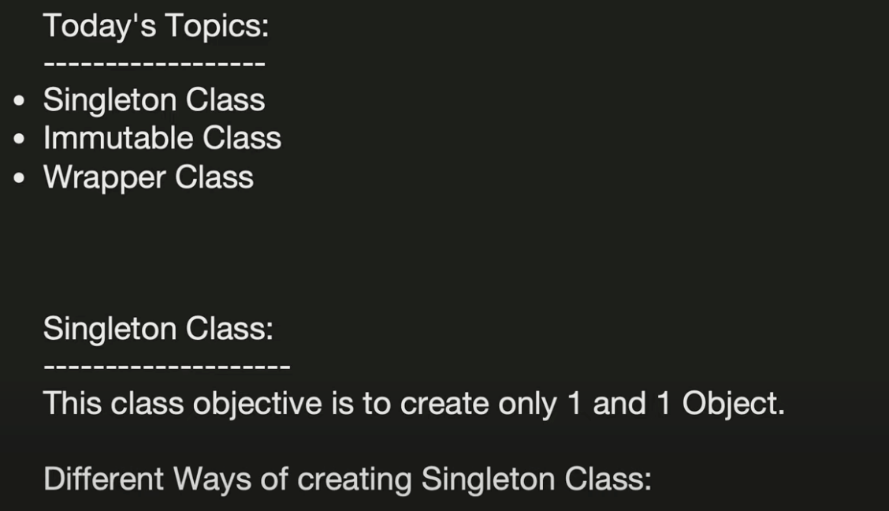
field.setAccessible(true) internally gives us the access of private fields outside the class and we can set its value. This is also disadvantage of reflection.

* Reflection breaks singleton class, and oops concept of encapsulation
* It is slow as it happens at runtime.

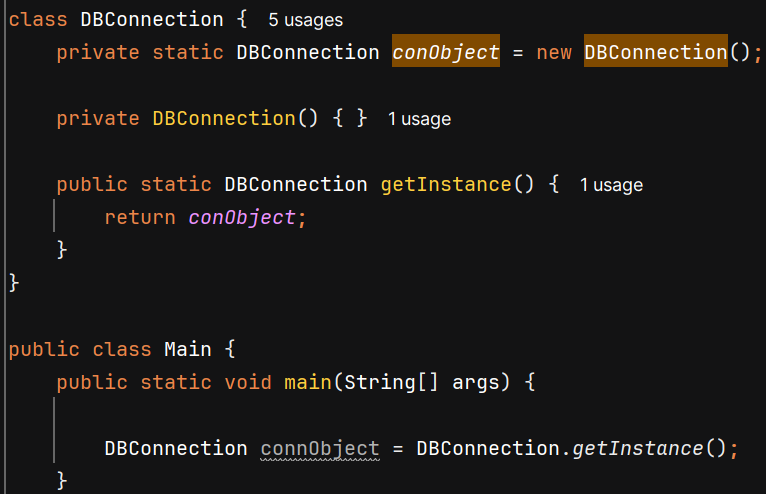
**Reflection of Constructor**



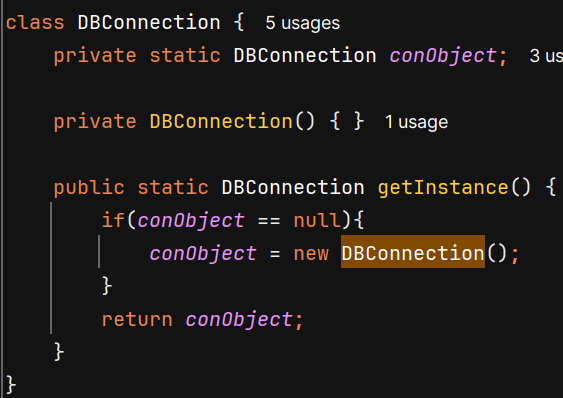


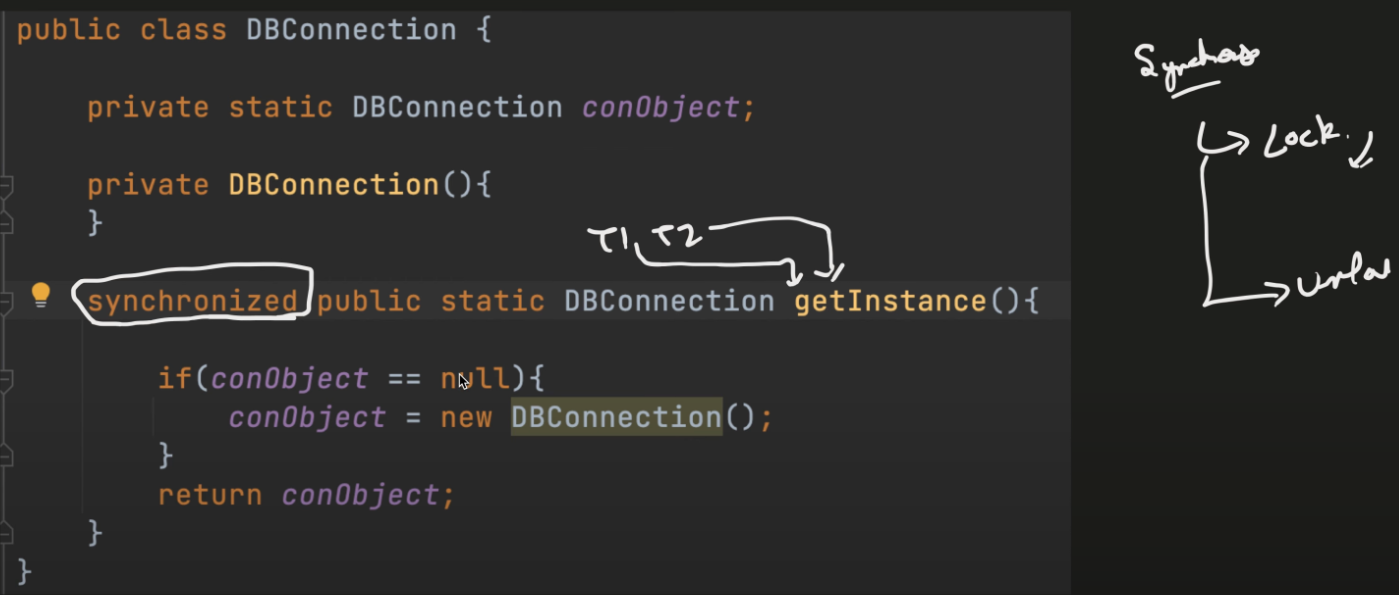


**Eager Initialization –**



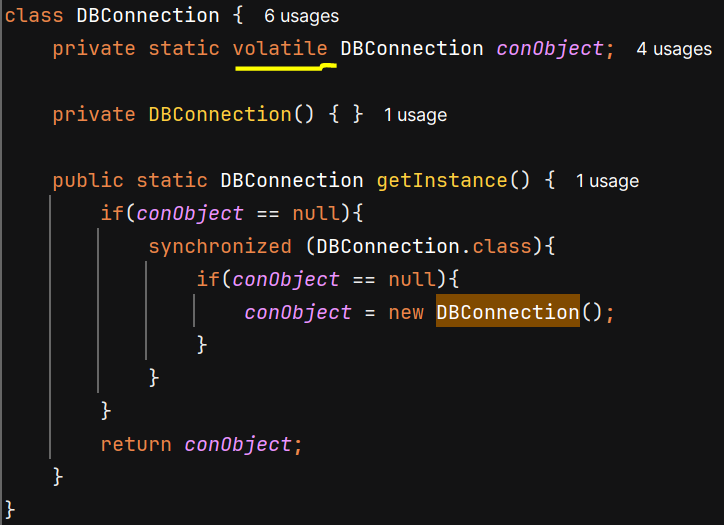
Static fields get initialized as soon as the class is loaded into memory.

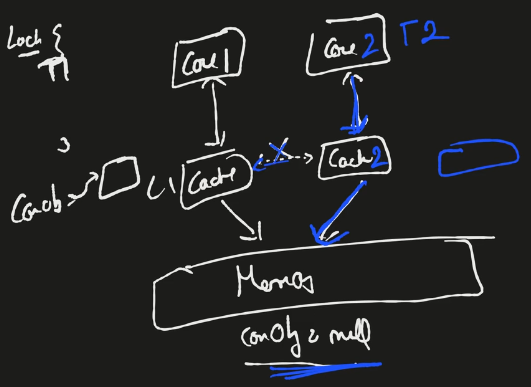
**Lazy Initialization –** 

**Synchronized Method –**

Problem: when you declare the entire method as synchronized, every call to getInstance() requires acquiring a lock, even after the instance has already been created.

**Double Locking –**



****

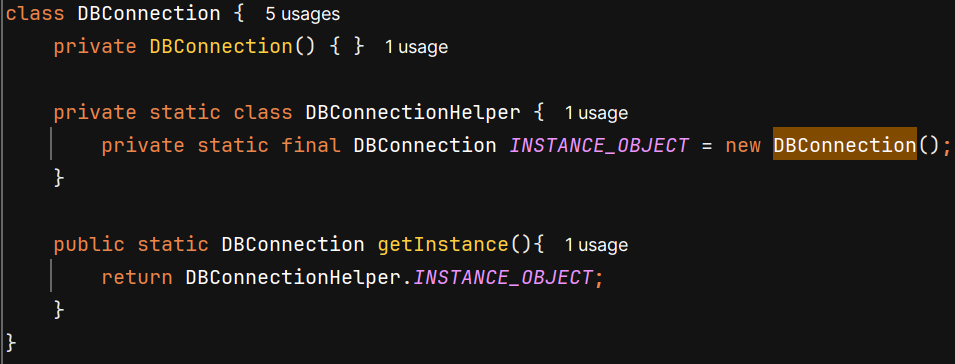
In a CPU, there are multiple core each having its own L1 cache and each CPU core may keep writes in its local cache and periodically update the main memory (RAM)

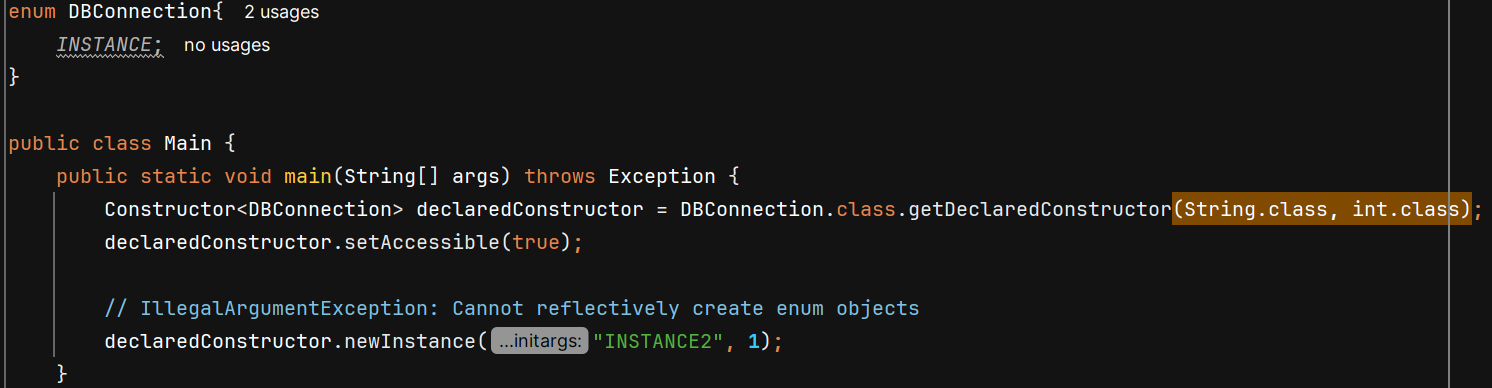
Suppose both thread1 and thread2 comes parallely. T1 starts running on core1 and T2 starts running on core2. T1 updates value of conObject but changes are only its cache, and sync up between core1 and core2 caches has not happened yet. Now T2 processing in core2 will see that conObject is still null so it will also create a new object. So, due to cache 2 objects are created and singleton class is not achieved.

**Bill Pugh Solution –**

It put initialization part in nested class as nested (static inner) class is not loaded / *initialized* when the outer class is loaded. JVM loads the inner class only when it is actively used.

JVM uses monitor lock to ensue that class initialization is thread-safe and gaurantees that a class’s static initialization is executed only once by a single thread.



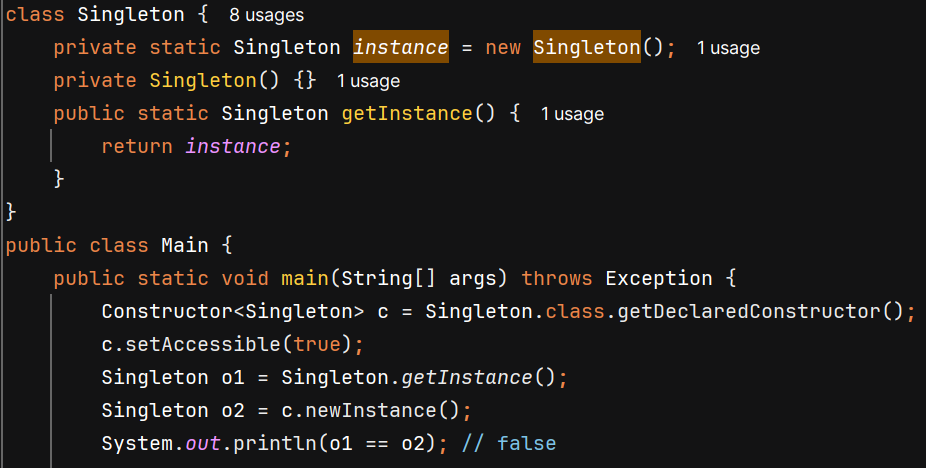
**ENUM –**

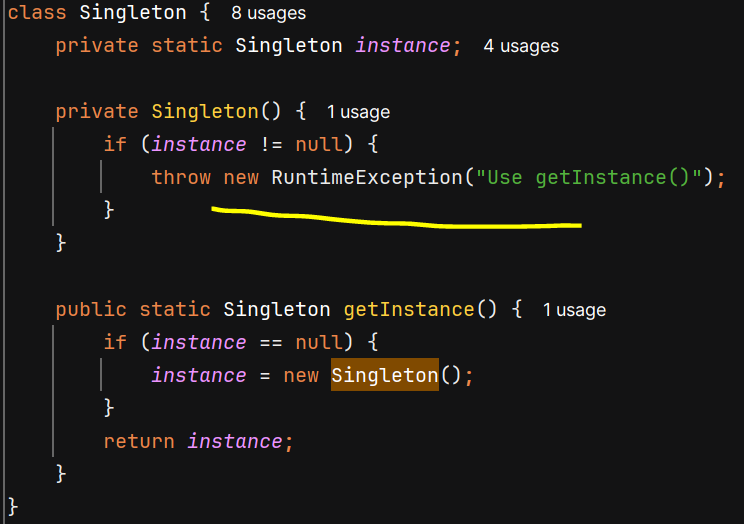
Enum guarantees that there will be only one instance for each defined constant, and you cannot create more instances of DBConnection because Enum don’t allow external instantiation.

Enum Singleton is safe from Reflection (get exception on newInstance()), Serialization (same instance is returned) and Cloning (cannot be cloned)

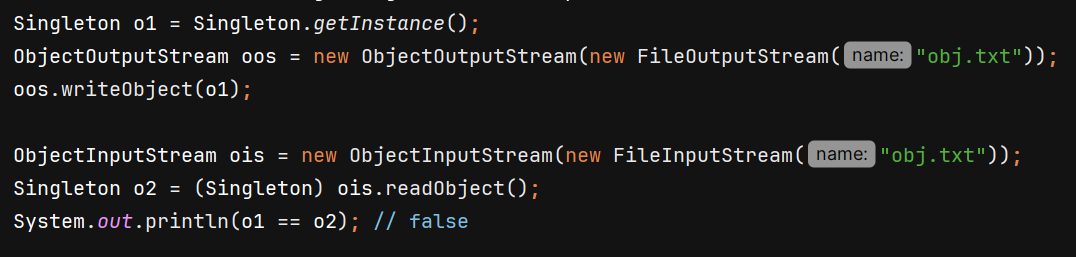
**Different ways to break Singleton and their solution**

**1. Using Reflection –**

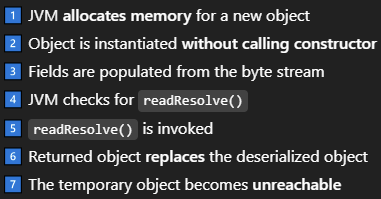
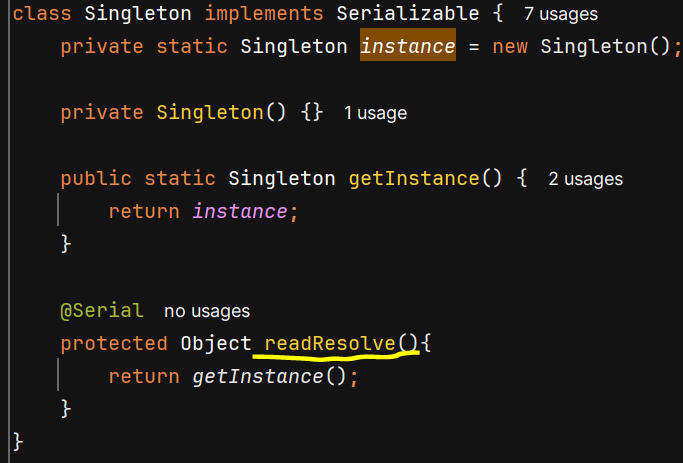


**Solution –** 

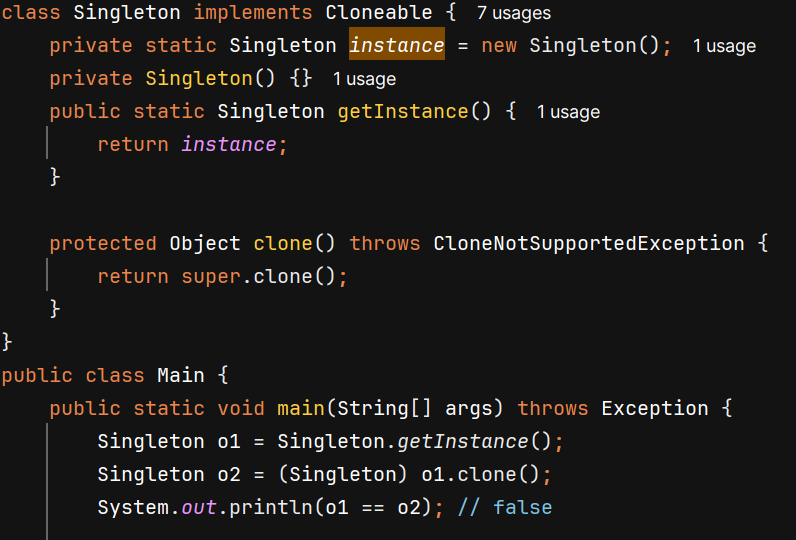
**2. Using Serialization –**



**Solution –**



**3. Using Clone –**



**Solution –** 