Parallel and MultiThreaded Programming

CSYE 7215

Homework 4

Due: October 11, 2020

Put all your java, compiled class files and documentation into a zip file named Homework4.zip and submit it via the dropbox on Canvas before the END of due date. Put your name on all .java files. There will be a short quiz on this assignment.

1. Explain every element in this JVM architecture, and how element layers are tied to each other?

Graphical user interface

Description automatically generated

1. Firstly Java compiler will compile the source code and generate the .class file which is further send to JVM
2. Once the JVM receive this file it perform following steps
   1. It will load the class file in main memory which is know as dynamic class loading using following class loader
      1. Bootstrap classloader
         * + Responsible for load the class from bootstrap classpath
      2. Extension classloader
         * + Responsible for load the classes from ext folder
      3. Application classloader
         * + Responsible for load application level classpath
   2. Inside the linking phase JVM verify the weather the generated bytecode is proper or not then allocate the memory for static variables and resolve the symbolic memory reference with original reference from memory area.
3. After all this process Run time data area comes into picture. It is divided into five major parts .
   1. **Method area** :
      1. All the class-level data will be stored here, including static variables.
      2. There is only one method area per JVM, and it is a shared resourcw
   2. **Heap Area** –
      1. All the Objects and their corresponding instance variables and arrays will be stored here.
      2. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe
   3. **Stack Area** –
      1. For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame.
      2. All local variables will be created in the stack memory
   4. **PC Registers** –
      1. Each thread will have separate PC Registers, to hold the address of current executing instruction
      2. Once the instruction is executed the PC register will be updated with the next instruction.
   5. **Native Method stack**s – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.
4. Once the bytecode assigned to run time data area , execution engine start executing bytecodes. JVM execution engine contains following components
   1. **Interpreter** –
      1. The interpreter interprets the bytecode faster but executes slowly.
      2. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
   2. **JIT Compiler** –
      1. The JIT Compiler neutralizes the disadvantage of the interpreter.
      2. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code.
      3. This native code will be used directly for repeated method calls, which improve the performance of the system.
   3. **Garbage Collector**:
      1. Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed.
      2. Garbage collection of the JVM collects the objects that are created.
5. JVM **also contains JNI and native method library**. **JNI will interact with native method library and provides the native library required for the** execution.

2. Explain:

**Implicit lock versus Explicit lock**

-> Java provides a built-in locking mechanism for enforcing atomicity: the synchronized block.

-> Every Java object can implicitly act as a lock for purposes of synchronization; these built-in locks are called intrinsic locks or monitor locks.

**->** The lock is automatically acquired by the executing thread before entering a synchronized block and automatically released when control exits the synchronized block, whether by the normal control path or by throwing an exception out of the block.

-> The only way to acquire an intrinsic lock is to enter a synchronized block or method guarded by that lock

*synchronized (someObject) {*

*// do some stuff*

*}*

*someObject.lock.acquire();*

*try {*

*// do some stuff*

*} finally {*

*someObject.lock.release();*

*}*

-> On the other hand Java has Lock interface which allows to implement abstract locking operations.

->Unlike Implicit locking, Lock offers a choice of unconditional, polled, timed, and interruptible lock acquisition, and all lock and unlock operations are explicit. Lock implementations must provide the same memory

*Lock lock = new ReentrantLock();*

*...*

*lock.lock(); try {*

*// update object state*

*// catch exceptions and restore invariants if necessary*

*} finally {*

*lock.unlock(); }*

* main difference between implicit lock and explicit lock

that it is not possible to interrupt a thread waiting to acquire a

lock, or to attempt to acquire a lock without being willing to wait for it forever

* Having a timeout trying to get access to a synchronized block is not possible. Using Lock.tryLock(long timeout, TimeUnit timeUnit), it is possible.
* The synchronized block must be fully contained within a single method. A Lock can have it’s calls to lock() and unlock() in separate methods

**Class lock versus Object lock**

**Class level Lock:**

* Class level lock prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime.
* Class level locking should always be done to make static data thread safe. As we know that static keyword associate data of methods to class level, so use locking at static fields or methods to make it on class level

*//Acquire lock on .class reference*

***synchronized (DemoClass.class)***

***{***

***//other thread safe code***

***}***

*//Lock object is static*

*private final static Object lock = new Object();*

***synchronized (lock)***

***{***

***//other thread safe code***

***}***

**Object level Lock:**

* Object level lock is mechanism when we want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe.

*// Using this*

***synchronized (this)***

***{***

***//other thread safe code***

***}***

OR

*// using private object instance*

*private final Object lock = new Object();*

*public void test(){*

***synchronized (lock)***

***{***

***//other thread safe code***

***}***

*}*

**Call Stack**

* The call stack is what a program uses to keep track of method calls. The call stack is made up of stack frames—one for each method call
* Each thread has it's own call stack. Let’s consider the following example of main thread

*Public class example {*

*Public static void main(String args[]){*

*Method1();*

*Method2();*

*Method3();*

*Method4();*

*}*

*}*

So call stack for aforementioned program would be like this

|  |
| --- |
| Method4() |
| Method3() |
| Method2() |
| Method1() |
| Main() |

**Stack Memory**

-> Java Stack memory is used for execution of a thread and it contains method specific values and references to other objects in Heap.

-> For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory.

-> The stack area is thread-safe since it is not a shared resource.

-> The Stack Frame is divided into three subentities:

Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.

Operand stack – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.

Frame data – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.

**Heap Space**

* All the Objects and their corresponding instance variables and arrays will be stored in heap space.
* There is one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe
* JVM heap is divided into 2 parts

Young generation - This is reserved for containing newly allocated objects. Young Gen includes three parts — **Eden Memory and two Survivor Memory spaces (S0, S1).** Most of the **newly** **created** objects **goes Eden space**. When Eden space is filled with objects, **Minor GC** is performed, and all the survivor objects are moved to one of the survivor spaces. Minor GC also checks the survivor objects and move them to the other survivor space. So, at a time, one of the survivor spaces is always empty.

Old Generation- This is reserved for containing long lived objects that could survive after many rounds of Minor GC. When Old Gen space is full, Major GC (a.k.a. Old Collection) is performed (usually takes longer time)

**String Pool, give example**

* String Pool is a storage area in Java heap. The JVM performs some steps while initializing string literals to increase performance and decrease memory overhead. To decrease the number of String objects created in the JVM, the String class keeps a pool of strings.
* Each time a string literal is created, the JVM checks the string literal pool first. If the string already exists in the string pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object initializes and is placed in the pool

Example

String s1 = “Hello”

String s2 = “Hello”

String s3 = “Test”

String s4 = new String(“test2”)

Heap Area

|  |
| --- |
| Hello |
| test |
|  |

S4

**Deadlock, Starvation, Race condition, provide examples**

Deadlock - Deadlock describes a situation where two or more

threads are blocked forever, waiting for each other

Example

* In Above example process p1 try to acquire lock on resource 1 but resource 1 is already locked by process p2 and process p2 try to acquire lock on resource r2 but it already locked by process p1. Now both the processes p1 and p2 are interdependent. This kind of situation known as Deadlock.

Starvation – Starvation describes a situation where a **thread is unable to gain regular access to shared resources and** is unable to make progress. This happens when shared resources are made **unavailable for long periods by "greedy" threads**

* For example, 5 thread running concurrently and for each thread we are setting some priority using **setPriority().** Method. If we are setting very high priority for three threads and low priority for 2 threads, then execution of low priority thread barely occurs and they hardly get chance to access the shared resources.

Race Condition - A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Execution order of threads are dependent on Thread scheduling algorithm, you don’t know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are “racing” to access/change the data

For Example Thread A reads variable count shared object into its CPU cache, and Thread B does the same but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache. If these increments had been carried out sequentially, the variable count would be incremented twice and had the original value + 2 written back to main memory. However, the two increments have been carried out concurrently without proper synchronization. Regardless of which thread A and B that writes its updated version of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments. Please find the reference code

*Public class RaceCondition implements Runnable{*

*Private int count= 0;*

*@override*

*Public void run(){*

*Count++;*

*}*

*}*

**How does Garbage collector works in Java?**

* In Java every time we create a new object it will acquire some space in JVM heap area.Garbage collector will help us to reclaimed the unused memory
* So basically when we creates a new object it will occupy the some space in JVM heap area and after **once that object is no longer referenced by the program**, **the heap space it occupies can be recycled so that the space is made available for subsequent new objects**.
* The garbage collector **must somehow determine which objects are no longer referenced by the program and make available the heap space occupied by such unreferenced objects**.
* In the process of freeing unreferenced objects, the garbage collector must run any finalizers of objects being freed.

**How does JVM manages garbage collector?**

* Whenever we are creating new object JVM allocate separate space to that object inside the heap memory area.
* Now JVM heap space is divided into two parts

1) Young Generation

2) Old Generation

Diagram, timeline

Description automatically generated

* The Young Generation is a part of the heap reserved for the allocation of new objects. When the nursery becomes full, garbage is collected by running a special young collection, where all the objects that have lived long enough in the nursery are promoted (moved) to the old space, thus freeing up the nursery for more object allocation. This garbage collection is called Minor GC. The Young Generation is divided 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 GC is performed and all the survivor objects are moved to one of the survivor spaces Minor GC also checks the survivor objects and moves them to the other survivor space. So at a time, one of the survivor space is always empty Objects that have survived many cycles of GC, are moved to the old generation memory space.**
* **When the old generation becomes full, garbage is collected there and the process is called as old collection. Old generation memory contains the objects that are long-lived and survived after many rounds of Minor GC. Usually, garbage collection is performed in Old generation memory when it’s full. Old generation garbage collection is called as “Major GC” and usually takes longer.**
* **The reasoning behind a Young Generation is that 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 nursery. Typically, a young collection frees a given amount of memory much faster than an old collection or a garbage collection of a single-generational heap (a heap without a nursery).**

3. Search internet to find all Threadsafe and NotThreadsafe Java HashMap,

HashTable, collections. For each collection:

**a) List three methods in each collection describing what is the intent of collection,**

**when it is useful to use this collection, and describe the selected methods.**

Collection are useful when we are working with the same type of objects

*Thread Safe classes in java collection frameworks*

- Stack

- Vector

- HashTable

- Blocking Queue

- ConcurrentMap

- ConcurrentNavigableMap

**Stack**

**push(E item)**🡪 Pushes an item onto the top of this stack.

**pop()**🡪Removes the object at the top of this stack and returns that object as the value of this function.

**peek()**🡪 Looks at the object at the top of this stack without removing it from the stack.

**Vector**

**add(E e)🡪**Appends the specified element to the end of this Vector.

**firstElement()🡪**Returns the first component (the item at index 0) of this vector.

**size()🡪**Returns the number of components in this vector.

**HashTable**

**clear()**🡪Clears this hashtable so that it contains no keys.

**get(Object key)🡪**Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

**remove(Object key)🡪**Removes the key (and its corresponding value) from this hashtable.

**Blocking Queue**

**put(E e)🡪**Inserts the specified element into this queue, waiting if necessary for space to become available.

**contains(Object o)🡪**Returns true if this queue contains the specified element.

**take()🡪**Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

**ConcurrentHashMap**

**elements()🡪**Returns an enumeration of the values in this table.

**keySet()**🡪Returns a Set view of the keys contained in this map.

**size()🡪**Returns the number of key-value mappings in this map.

*Not Thread safe*

- ArrayList

- HashMap

- LinkedList

- HashSet

- Linked hashset

- Tree Hashset

**ArrayList**

**isEmpty()🡪**Returns true if this list contains no elements.

**contains(Object o)🡪**Returns true if this list contains the specified element.

**clear()🡪**Removes all of the elements from this list.

**HashMap**

**put(K key, V value)🡪**Associates the specified value with the specified key in this map.

**remove(Object key)🡪**Removes the mapping for the specified key from this map if present.

**containsKey(Object key)🡪**Returns true if this map contains a mapping for the specified key.

**LinkedList**

**getFirst()🡪**Returns the first element in this list.

**peekFirst()🡪**Retrieves, but does not remove, the first element of this list, or returns null if this list is empty.

**set(int index, E element)🡪**Replaces the element at the specified position in this list with the specified element.

**HashSet**

**contains(Object o)🡪**Returns true if this set contains the specified element.

**iterator()🡪**Returns an iterator over the elements in this set.

**add(E e)🡪**Adds the specified element to this set if it is not already present.

**LinkedhashSet**

**remove(Object o)🡪**Removes the specified element from this set if it is present.

**size()🡪**Returns the number of elements in this set (its cardinality).

**clear()🡪**Removes all of the elements from this set.

**b) What is Collections class? Most methods in this class are static, name five meth-ods.**

Java collections class

- This class consists exclusively of static methods that operate on or return collections

- class contains polymorphic algorithms that operates on collection

known as "Wrapper"

**Class contains three static field**

List🡪The empty list (immutable). This list is serializable.

map 🡪 The empty map (immutable). This map is serializable.

set 🡪 The empty set (immutable). This set is serializable.

Collections class contains lots of static methods

***Useful methods in Collections class***

**1) public static <T> Collection<T> synchronizedCollection(Collection<T> c)**

Returns a synchronized (thread-safe) collection backed by the specified collection. In order to guarantee serial access, it is critical that all access to the backing collection is accomplished through the returned collection.

example:

Collection c = Collections.synchronizedCollection(myCollection);

...

synchronized (c) {

Iterator i = c.iterator(); // Must be in the synchronized block

while (i.hasNext())

foo(i.next());

}

**2) public static <T> Set<T> synchronizedSet(Set<T> s)**

Returns a synchronized (thread-safe) set backed by the specified set.

example

Set s = Collections.synchronizedSet(new HashSet());

...

synchronized (s) {

Iterator i = s.iterator(); // Must be in the synchronized block

while (i.hasNext())

foo(i.next());

}

**3) public static <T> List<T> synchronizedList(List<T> list)**

Returns a synchronized (thread-safe) list backed by the specified list.

example :

List list = Collections.synchronizedList(new ArrayList());

...

synchronized (list) {

Iterator i = list.iterator(); // Must be in synchronized block

while (i.hasNext())

foo(i.next());

}

**4) public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)**

Returns a synchronized (thread-safe) map backed by the specified map.

example :

Map m = Collections.synchronizedMap(new HashMap());

...

Set s = m.keySet(); // Needn't be in synchronized block

...

synchronized (m) { // Synchronizing on m, not s!

Iterator i = s.iterator(); // Must be in synchronized block

while (i.hasNext())

foo(i.next());

}

**5) public static <T> void sort(List<T> list,**

**Comparator<? super T> c)**

It is used to sort the elements present in the specified list of Collection in ascending order

example :

ArrayList<String> al = new ArrayList<String>();

/\* Collections.sort method is sorting the

elements of ArrayList in ascending order. \*/

Collections.sort(al)

4. In Homework3, you created 50 student threads and one GraderThread. Change the program to use Explicit locking instead of implicit locking. Note: see problem descrip-tion in hw3, problem-3 (a) (b) (c) (d), and all the requirements for that problem must be implemented in this problem using explicit locking.

Approach :

🡪 As we have already implemented the logic of the give program using synchronized block now I am eliminating synchronized block (implicit locking). Instead of that **I’ve used the ReentrantLock lock class which implements the lock interface**

🡪 To replace the notifyAll and wait method I have used the condition class implementation in which we have methods such as **await()** and **signalAll()** which performs the similar task.

🡪 Please find the following output screenshot using Implicit locking. To review detail implementation please check attached solution file

A close up of a computer

Description automatically generated

Text

Description automatically generated

**5. Design a program that creates 10 Student objects with each student is size of 20 bytes. The JVM heap size is 240 bytes. Your program design should consider scenario where five objects become Unreferenced for garbage collection. Provide the design cri-terion for Minor GC and Major GC. You should provide the detail design for JVM managing garbage collector memory management.**

- The JVM heap size determines how often and how long the VM spends collecting garbage. An acceptable rate for garbage collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections.

- If **we set a large heap size, full garbage collection is slower**, but it **occurs less frequently. If** you set **your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently.**

->In our application we know that we need to set the Heap size to 240 bytes. So we can specify those setting using following command.

*java -Xms240m -Xmx240m -XX:NewRatio=2 -XX:SurvivorRatio=12*

-XmsSetting — initial Heap size

-XmxSetting — maximum Heap size

-XX:NewRatio – specify the ration between new and old generation

-XX:SurvivorRatio - ratio between eden and a survivor space

|  |  |  |  |
| --- | --- | --- | --- |
| <----------------------------240 bytes-----------------------------------------------🡪 | | | |
| ----------------120 bytes--------- | | | ----------------120 bytes--------- |
| Young Generation | | |  |
| Eden Space  *100 Bytes* | S1  *10 Bytes* | S2  *10 Bytes* | Old generation |
| New Student object will resides here |  |  | old student object will resides here |

Explanation

* Here I have specify the ratio between young and old generation is ½ means 120 byte will be allocated to each generation
* The ration between Eden and survivor space is 1/12 means each survivor space will get 10 bytes and Eden space will get 100 bytes.
* **We know that Eden space is the area where newly created object resides and once it will gets full Minor GC will perform and reclaimed the unused memory and move the survivor objects to survivor space so in our application student object will occupy 20 bytes so if we creates the 5 student objects then Eden space area gets full and Minor GC will perform.**
* **To makes sure that Minor GC perform more frequently I have set the Eden space size to 100 bytes for robust performance**
* **If I will increase the younger generation size then the minor GC will perform in less frequency and when Older Generation gets full Major GC need to do more work.**

Sample Code:

Public class student {// ….};

Public class testGC {

Student s1 = new Student();

Student s2 = new Student();

Student s3 = new Student();

s3 = null // unreferenced the s3

Student s4 = new Student();

Student s5 = new Student();

Student s6 = new Student();

**// eden space gets full and Mino GC performed and remove reclaimed s3 memory**

}

6. Synchronized blocks in Java are Reentrant. That is if a Java thread enters a syn-chronized block code, and thereby takes the lock on the monitor object the block is synchronized on, the thread can enter other Java code blocks synchronized on the same monitor object? a) what is object monitor in this code? b) Explain how the fol-lowing code works?

**c) Provide different Test scenarios that can successfully execute this code.**

public class Reentrant {

int level = 0;

Lock lock = new Lock();

public outer() {

lock.lock();

inner();

lock.unlock();

}

public synchronized inner(){

level++;

lock.lock();

try {

if (level <= 3) {

inner();

if (level == 2) {

Thread.sleep(1000);

}

} else {

Thread.sleep(1000);

}

} finally {

lock.unlock();

level--;

}

} }

* If we are creating instance of Lock we will get compilation error as Lock is an interface we need to create the instance of class which implements the Lock interface

Error snap:

Graphical user interface, text

Description automatically generated

* To resolved the above error I have now used Reentrant class instance which implements the Lock Interface.

*Lock lock = new ReentrantLock();*

* Now we have two different method in reentrant class one is outer() and one is inner(). Outer method acquire the explicit lock and call the inner method and then inner method will recursively call itself and increment the level value till level 4 and then decrement the value in backtracking till 0 in finally block.
* The **Instance of RentrantLock class will work as object monitor here**
* **Here we can observer the Reentrant behavior of ReentrantLock class , thread is acquiring the lock on lock object multiple times and it allows us to hold the lock on same lock instance.**
* **Even if we will call the inner method directly from main class it will same output. Inner method is recursively call itself and acquiring the lock on lock instance in each call and then release the lock during the backtracking.**

**Output: calling inner method directly**

**Text

Description automatically generated**

**Output: calling outer method directly**

**Graphical user interface, text

Description automatically generated**

7. Consider the following code using Reentrance Lock. How does the Lock work in the following program. Compile and Run.

class PrintingJob implements Runnable

{

private PrinterQueue printerQueue;

public PrintingJob(PrinterQueue printerQueue)

{

this.printerQueue = printerQueue;

}

@Override

public void run()

{

System.out.printf("%s: Going to print a document\n”,

Thread.currentThread().getName());

printerQueue.printJob(new Object());

}

}

class PrinterQueue

{

private final Lock queueLock = new ReentrantLock();

public void printJob(Object document)

{

queueLock.lock();

try

{

Long duration = (long) (Math.random() \* 10000);

System.out.println(Thread.currentThread().getName() +

": PrintQueue: Printing a Job during " + (duration / 1000) +

" seconds :: Time - " + new Date());

Thread.sleep(duration);

} catch (InterruptedException e)

{

e.printStackTrace();

} finally

{

System.out.printf("%s: The document has been printed\n”,

Thread.currentThread().getName());

queueLock.unlock(); }

} }

public class LockExample

{

public static void main(String[] args)

{

PrinterQueue printerQueue = new PrinterQueue();

Thread thread[] = new Thread[10];

for (int i = 0; i < 10; i++)

{

thread[i] = new Thread(new PrintingJob(printerQueue), "Thread " + i);

}

for (int i = 0; i < 10; i++)

{

thread[i].start();

}

}

}

Explanation:

In given program we are creating two different classes 1 form **printingJob** and one is **Printerqueue**. **PrintingJob** class implements the runnable interface and call the **printJob** method of **printerqueue** class.

To handle multiple printing job we have used the explicit lock using **ReentarntLock** class (which is implements the Lock interface)

Inside the printJob method we are. acquiring explicit lock using **lock() method and releasing that lock in finally block using unlock() method.**

This locking mechanism make sure that at a time only one thread can execute the critical part of method. We can implement same thing **using synchronized block as well.**

Output :

Graphical user interface, text

Description automatically generated

8. Consider the following class Customer class, Build a diagram to show

Call Stack, Memory Stack, and Heap.

public class Customer{

private String name;

private String address;

private String vehicle;

private double amount;

private double charge;

private double TotalAmount;

private int CustomerCount;

public Customer() {

}

public Customer(name, address, vehicle, amount,

charge, TotalAmount, CustomerCount) {

this.name=name;

this.address=address;

this.vehicle=vehicle;

this.amount=amount;

this.charge=charge;

this.TotalAmount=TotalAmount;

this.CustomerCount=CustomerCount;

}

public String getName()

{

return name;

}

public String getAddress()

{

return address;

}

public String getVehicle()

{

return vehicle;

}

public double getAmount()

{

return amount;

}

public double getCharge()

{

return charge;

}

public double getTotalAmount()

{

return TotalAmount;

}

public int getCustomerCount()

{

return CustomerCount;

}

}

public class TestDriver {

public static void main(String[] args) {

private String m\_name=“John”;

private String m\_address=“123 XYZ”;

private String m\_vehicle=“Toyota”;

private double m\_amount=20;

private double m\_charge=25;

private double m\_TotalAmount=140;

private int m\_CustomerCount=10;

Customer c = new Customer(m\_name, m\_address, m\_;

c = new Customer(m\_name, m\_address, m\_vehicle,

m\_amount, m\_charge, m\_TotalAmount, m\_Customer\_Count);

}

}

Call Stack, Stack Frame and heap space interaction

Diagram

Description automatically generated

Explanation:

In Given Program call stack would me in this sequence

1. Main()
2. Customer()
3. Customer()

Call stack execute in LIFO manner means last in First out

* In Stack frame each frame contains the value of all the local variable such as m\_name, m\_address etc. Here string values are assign using literals hence all th local variable string value will point to **string pool**While we have double variable as well which will hold 2 consecutive entries in local variable arrays.
* Initially Customer object c assigned by new Customer object then same c object assigned by another new customer constructor hence the **previous assigned object space in heap will be reclaim by the garbage collector**