Memory Leak and Java Code

# Misuse of java code

Java implicitly reclaims memory by GC (a daemon thread). GC periodically checks, if there is any object which is unreachable or to be precise no reference pointing to the same.  If so GC reclaims it's acquired memory.

Now the question is should we worry about memory Leak or java handles it?

Pay attention to the definition,

***An Object is eligible for garbage collection when it is unreachable(unused), no living thread can reach it.***

So if an object which is not used in an application but unintentionally it has references same is nor eligible for garbage collection a potential Memory Leak.

***GC takes care of unreachable objects, but can’t determine unused objects, Unused Object depends on Application logic so coder must have to pay attention to the business code. Silly mistakes silently grew up as MONSTER.***

Memory leaks can occur in many ways, I will take some of the examples.

**Example 1** : **Autoboxing**

**package** com.example.memoryleak;

**public** **class** Adder {

**public** **long** addIncremental(**long** l)

{

Long sum=0L;

sum=sum+l;

**return** sum;

}

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

Adder adder = **new** Adder();

**long** totsum =0;

**for**(**long** i=0;i<1000;i++)

{

adder.addIncremental(i);

}

}

}

Can you spot the memory Leak?

Here, I made a mistake instead of taking primitive long for the sum I took Long(wrapper class) which is the cause of memory leak.  Due to auto-boxing, sum**=sum+l;** creates new Object in every iteration so 1000 unnecessary Objects will be created. Please avoid mix and match between primitives and wrapper class. Try to use primitive as long as you can.

**Example 2: Using cache**

**package** com.example.memoryleak;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** Cache {

**private** Map<String,String> map= **new** HashMap<String,String>();

**public** **void** initCache()

{

map.put("Anil", "Work as Engineer");

map.put("Shamik", "Work as Java Engineer");

map.put("Ram", "Work as Doctor");

}

**public** Map<String,String> getCache()

{

**return** map;

}

**public** **void** forEachDisplay()

{

**for**(String key : map.keySet())

{

String val = map.get(key);

System.***out***.println(key + " :: "+ val);

}

}

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

Cache cache = **new** Cache();

cache.initCache();

cache.forEachDisplay();

}

}

Here memory leak occurs due to the internal map data structure. This class is for display employee value from the cache. Once those are displayed, further there is no need to store those elements in the cache.

We forgot to clear the cache so although objects in cache are not required anymore by the application, but it can’t be GCed as map holds a strong reference to them.

So when you using your own Cache don’t forget to clear them if items in the cache are no longer requires. Alternatively, you can initialize cache by WeakHashMap, the beauty of WeakHashMap is, if keys are not referenced by any other objects that entry will be eligible for garbage collection.

There is lot say about WeakHashMap I will discuss it in another article. But use same with caution, if you want to reuse the values stored in the cache, it may happen its key is not referenced by any other object so entry will be GCed and that value magically disappears.

**Example 3: Closing connection**

try

{

Connection con = DriverManager.getConnection();

…………………..

con.close();

}

Catch(exception ex)

{

}

In above example, we close the connection(Costly) resource in the try block, so in the case of an exception, the connection will not be closed. So it creates a memory leak as this connection never return back to the pool.

Please always put any closing stuff in the finally block.

**Example 4: Using CustomKey**

**package** com.example.memoryleak;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** CustomKey {

**public** CustomKey(String name)

{

**this**.name=name;

}

**private** String name;

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

Map<CustomKey,String> map = **new** HashMap<CustomKey,String>();

map.put(**new** CustomKey("Shamik"), "Shamik Mitra");

String val = map.get(**new** CustomKey("Shamik"));

System.***out***.println("Missing equals and hascode so value is not accessible from Map " + val);

}

}

As in CustomKey we forgot to provide equals() and hashcode() implementation so a key and value once stores in map can’t be retrieved later, as map get() method checks hashcode() and equals(). But this entry is not able to GCed as the map has a reference to it but application can’t access it. Surely a memory leak.

So when make your Custom key always provide a equals and hashcode() implementation.

**Example 5**: **Mutable Custom Key**

**package** com.example.memoryleak;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** MutableCustomKey {

**public** MutableCustomKey(String name)

{

**this**.name=name;

}

**private** String name;

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + ((name == **null**) ? 0 : name.hashCode());

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

MutableCustomKey other = (MutableCustomKey) obj;

**if** (name == **null**) {

**if** (other.name != **null**)

**return** **false**;

} **else** **if** (!name.equals(other.name))

**return** **false**;

**return** **true**;

}

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

MutableCustomKey key = **new** MutableCustomKey("Shamik");

Map<MutableCustomKey,String> map = **new** HashMap<MutableCustomKey,String>();

map.put(key, "Shamik Mitra");

MutableCustomKey refKey = **new** MutableCustomKey("Shamik");

String val = map.get(refKey);

System.***out***.println("Value Found " + val);

key.setName("Bubun");

String val1 = map.get(refKey);

System.***out***.println("Due to MutableKey value not found " + val1);

}

}

Although here we provide equals() and hashcode(), for custom Key but we make it mutable so if unintentionally after storing same into the map, if it’s property is changed that entry will never found by the application but map hold a reference so memory leak happens.

Always make your custom key immutable.

**Example 6: Internal Data Structure**

**package** com.example.memoryleak;

**public** **class** Stack {

**private** **int** maxSize;

**private** **int**[] stackArray;

**private** **int** pointer;

**public** Stack(**int** s) {

maxSize = s;

stackArray = **new** **int**[maxSize];

pointer = -1;

}

**public** **void** push(**int** j) {

stackArray[++pointer] = j;

}

**public** **int** pop() {

**return** stackArray[pointer--];

}

**public** **int** peek() {

**return** stackArray[pointer];

}

**public** **boolean** isEmpty() {

**return** (pointer == -1);

}

**public** **boolean** isFull() {

**return** (pointer == maxSize - 1);

}

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

Stack stack = **new** Stack(1000);

**for**(**int** i=0;i<1000;i++)

{

stack.push(i);

}

**for**(**int** i=0;i<1000;i++)

{

**int** element = stack.pop();

System.***out***.println("Poped element is "+ element);

}

}

}

Here we face a tricky problem when stack first grows then shrinks. Actually, it is due to the internal implementation. Stack internally holds an array but for application perspective, the active portion of Stack is where pointer its current point.

So when stack grows to 1000, internally array cells are fill up with elements but afterwards when we popped all elements, pointer comes to zero so as per application it is empty but still internal array contains all popped reference. In java, we call it **Obsolete reference. An obsolete reference is such a reference which can’t be dereferenced.**

And this reference can’t be GCed as array holds those elements, but they are unnecessary after popping.

To fix it , we need to set null value when the pop action occurs so those objects are able to garbage collection.

**public** **int** pop() {

int size = pointer--

**int element=** stackArray[size];

stackArray[size]=null;

return element;

}

***Safety measure for preventing memory Leak:***

**Conclusion:**Due to silly mistakes memory leaks happen so when coding please review your code to prevent the memory leak**.**