



[Calling All Contributors] Get Rewarded for Your Writing

[Visit the Bounty Board](#) ▶

Memory Leaks and Java Code

by Shamik Mitra MVB · Aug. 29, 16 · Java Zone

Just released, a free O'Reilly book on Reactive Microsystems: The Evolution of Microservices at Scale. Brought to you in partnership with Lightbend.

Java implicitly reclaims memory by GC (a daemon thread). GC periodically checks if there is any object which is unreachable or, to be precise, has no reference pointing to that object. If so, GC reclaims the newly-available memory.

Now the question is should we worry about memory leaks or how Java handles it?

Pay attention to the definition: ***An object is eligible for garbage collection when it is unreachable (unused), and no living thread can reach it.***

So if an object which is not used in an application but unintentionally has references, it is not eligible for garbage collection, and is a potential memory leak.

GC takes care of unreachable objects, but can't determine unused objects. Unused objects depend on application logic, so a programmer must pay attention to the business code. Silly mistakes silently grow up to be a monster.

Memory leaks can occur in many ways, I will look at some examples.

Example 1: Autoboxing

```
1 package com.example.memoryleak;
2 public class Adder {
3     public long addIncremental(long l)
4     {
5         Long sum=0L;
6         sum =sum+l;
7         return sum;
8     }
9     public static void main(String[] args) {
10         Adder adder = new Adder();
11         for(long i;i<1000;i++)
12         {
13             adder.addIncremental(i);
14         }
15     }
```

```
16 }
```

Can you spot the memory leak?

Here I made a mistake. Instead of taking the primitive long for the sum, I took the Long (wrapper class), which is the cause of the memory leak. Due to auto-boxing, `sum=sum+l;` creates a new object in every iteration, so 1000 unnecessary objects will be created. Please avoid mixing and matching between primitive and wrapper classes. Try to use primitive as much as you can.

Example 2: Using Cache

```
1 package com.example.memoryleak;
2 import java.util.HashMap;
3 import java.util.Map;
4 public class Cache {
5     private Map<String,String> map= new HashMap<String,String>();
6     public void initCache()
7     {
8         map.put("Anil", "Work as Engineer");
9         map.put("Shamik", "Work as Java Engineer");
10        map.put("Ram", "Work as Doctor");
11    }
12    public Map<String,String> getCache()
13    {
14        return map;
15    }
16    public void forEachDisplay()
17    {
18        for(String key : map.keySet())
19        {
20            String val = map.get(key);
21            System.out.println(key + " :: "+ val);
22        }
23    }
24    public static void main(String[] args) {
25        Cache cache = new Cache();
26        cache.initCache();
27        cache.forEachDisplay();
28    }
29 }
```

Here, a memory leak occurs due to the internal map data structure. This class is to display the employee value from the cache. Once those are displayed, 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, it can't be GCed, as map holds a strong reference to them.

So when you're using your own Cache, don't forget to clear them if items in the cache are no longer required. Alternatively, you can initialize cache by WeakHashMap. The beauty of WeakHashMap is, if keys are not

referenced by any other objects, then that entry will be eligible for GC.

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

Example 3: Closing Connections

```
1  try
2  {
3      Connection con = DriverManager.getConnection();
4      .....
5      con.close();
6  }
7
8  Catch(exception ex)
9  {
10 }
```

In the 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

```
1  package com.example.memoryleak;
2  import java.util.HashMap;
3  import java.util.Map;
4  public class CustomKey {
5      public CustomKey(String name)
6      {
7          this.name=name;
8      }
9      private String name;
10     publicstaticvoid main(String[] args) {
11         Map<CustomKey,String> map = new HashMap<CustomKey,String>();
12         map.put(new CustomKey("Shamik"), "Shamik Mitra");
13         String val = map.get(new CustomKey("Shamik"));
14         System.out.println("Missing equals and hascode so value is not acces
15     }
16 }
```

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

So when you make your Custom key, always provide an equals and hashCode() implementation.

Example 5: Mutable Custom Key

```
1 package com.example.memoryleak;
2 import java.util.HashMap;
3 import java.util.Map;
4 public class MutableCustomKey {
5     public MutableCustomKey(String name)
6     {
7         this.name=name;
8     }
9     private String name;
10    public String getName() {
11        return name;
12    }
13    public void setName(String name) {
14        this.name = name;
15    }
16    @Override
17    public int hashCode() {
18        final int prime = 31;
19        int result = 1;
20        result = prime * result + ((name == null) ? 0 : name.hashCode());
21        return result;
22    }
23    @Override
24    public boolean equals(Object obj) {
25        if (this == obj)
26            return true;
27        if (obj == null)
28            return false;
29        if (getClass() != obj.getClass())
30            return false;
31        MutableCustomKey other = (MutableCustomKey) obj;
32        if (name == null) {
33            if (other.name != null)
34                return false;
35        } elseif (!name.equals(other.name))
36            return false;
37        return true;
38    }
39    public static void main(String[] args) {
40        MutableCustomKey key = new MutableCustomKey("Shamik");
41        Map<MutableCustomKey,String> map = new HashMap<MutableCustomKey,String>();
42        map.put(key, "Shamik Mitra");
43        MutableCustomKey refKey = new MutableCustomKey("Shamik");
```

```
44         String val = map.get(refKey);
45         System.out.println("Value Found " + val);
46         key.setName("Bubun");
47         String val1 = map.get(refKey);
48         System.out.println("Due to MutableKey value not found " + val1);
49     }
50 }
```

Although here we provided equals() and hashCode() for the custom Key, we made it mutable unintentionally after storing it into the map. If its property is changed, then that entry will never be found by the application, but map holds a reference, so a memory leak happens.

Always make your custom key immutable.

Example 6: Internal Data Structure

```
1  package com.example.memoryleak;
2  public class Stack {
3      private int maxSize;
4      private int[] stackArray;
5      private int pointer;
6      public Stack(int s) {
7          maxSize = s;
8          stackArray = new int[maxSize];
9          pointer = -1;
10     }
11     public void push(int j) {
12         stackArray[++pointer] = j;
13     }
14     public int pop() {
15         return stackArray[pointer--];
16     }
17     public int peek() {
18         return stackArray[pointer];
19     }
20     public boolean isEmpty() {
21         return (pointer == -1);
22     }
23     public boolean isFull() {
24         return (pointer == maxSize - 1);
25     }
26     public static void main(String[] args) {
27         Stack stack = new Stack(1000);
28         for(int i=0;i<1000;i++)
29         {
30             stack.push(i);
31         }
32         for(int i=0;i<1000;i++)
```

```
33         {  
34             int element = stack.pop();  
35             System.out.println("Poped element is "+ element);  
36         }  
37     }  
38 }
```

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 from an application perspective, the active portion of Stack is where the pointer is pointing.

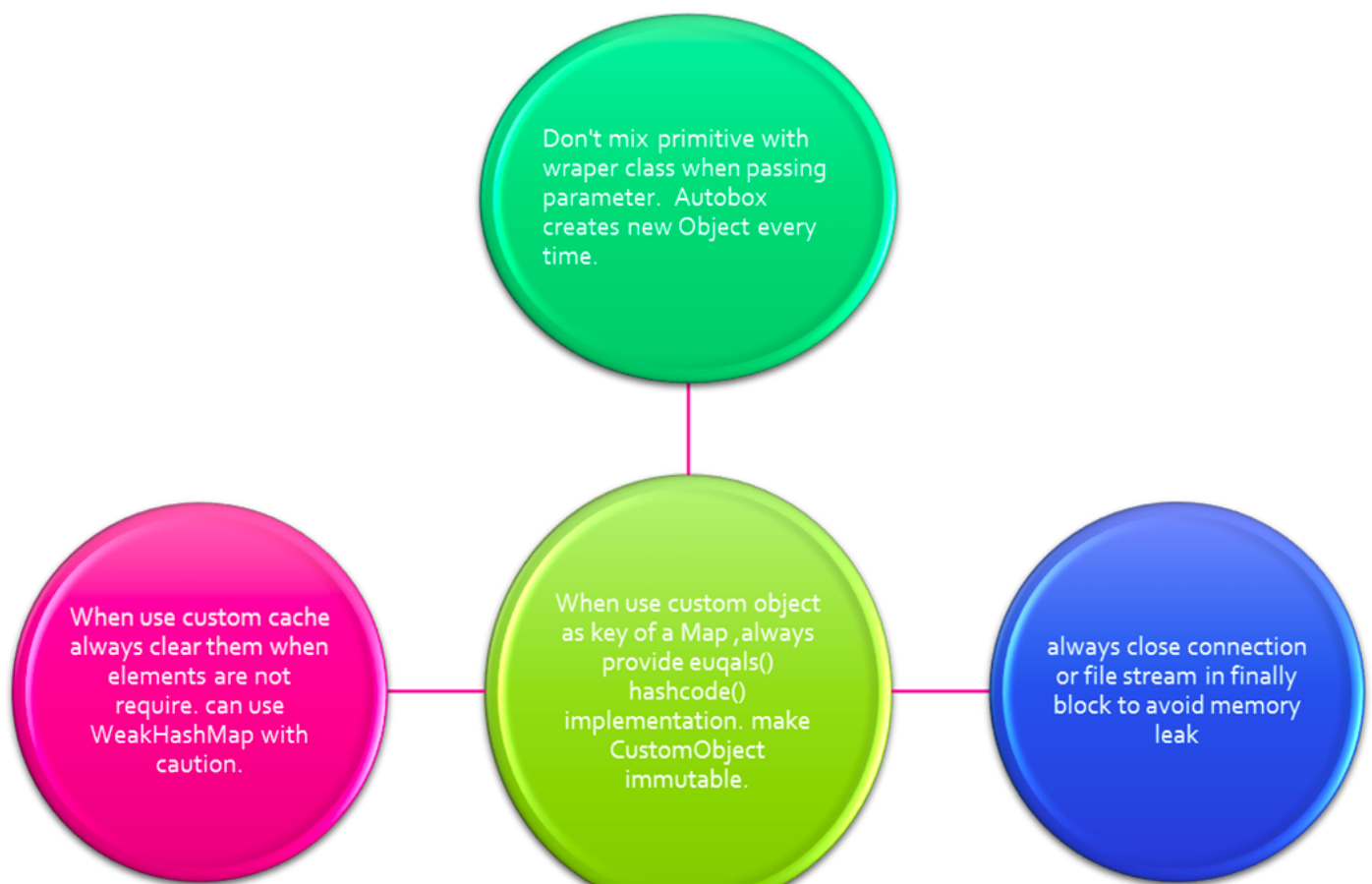
So when Stack grows to 1000, internally the array cells are filled up with elements, but afterwards when we pop all elements, the pointer comes to zero, so according to the application it is empty, but the internal array contains all popped references. In Java, we call it an **obsolete reference**. **An obsolete reference is a reference which can't be dereferenced.**

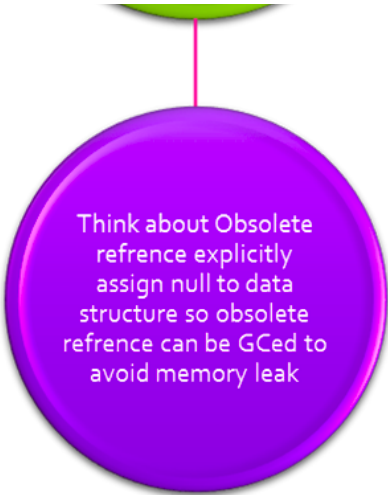
This reference can't be GCed, as the array holds those elements, but they are unnecessary after they are popped.

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

```
1 public int pop() {  
2     int size = pointer--  
3     int element= stackArray[size];  
4     stackArray[size];  
5     return element;  
6 }
```

Safety Measure for Preventing Memory Leaks:





Think about Obsolete
reference explicitly
assign null to data
structure so obsolete
reference can be GCed to
avoid memory leak

Strategies and techniques for building scalable and resilient microservices to refactor a monolithic application step-by-step, a free O'Reilly book. Brought to you in partnership with Lightbend.

Like This Article? Read More From DZone



**Potential Java Garbage Collection
Interview Questions**



**Java Memory Architecture Cheat
Sheet**




**Understanding the Java Memory
Model and Garbage Collection**



**Free DZone Refcard
Getting Started With Play
Framework**

Topics: GARBAGE COLECTION , MEMORY LEAK , JAVA

Published at DZone with permission of Shamik Mitra, DZone MVB. [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

Java Partner Resources

Reactive Microsystems - The Evolution of Microservices at Scale
Lightbend



Predictive Analytics + Big Data Quality: A Love Story
Melissa Data



Build vs Buy a Data Quality Solution: Which is Best for You?
Melissa Data



jQuery UI and Auto-Complete Address Entry
Melissa Data



