**What is Memory leak java**

* A Memory Leak is a situation **where there are objects present in the heap that are no longer used, but the garbage collector is unable to remove them from memory,** and therefore, they’re unnecessarily maintained.
* It is bad because it **blocks memory resources and degrades system performance over time**.
* If finally terminating with a fatal *java.lang.OutOfMemoryError not deal with it*.

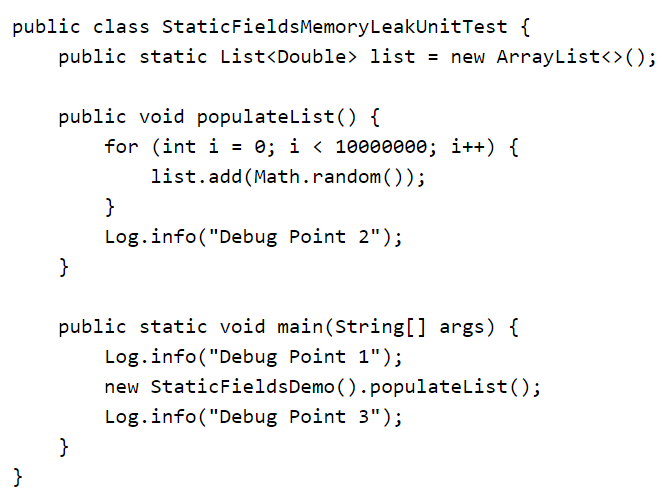
**Types of Memory Leaks in Java**

* **Memory Leak Through *static* Fields**
* **Through Unclosed Resources**
* **Improper *equals()* and*hashCode()* Implementations**
* **Inner Classes That Reference Outer Classes**
* **Through *finalize()* Methods**
* **Interned *Strings***

1. **Memory Leak Through *static* Fields**

* heavy use of *static* variables may cause the memory leak issue.
* In Java, *static*fields have a life that usually matches the entire lifetime of the running application

e.g



**How to Prevent It?**

* Minimize the use of static variables.
* When using singletons, rely upon an implementation that lazily loads the object, instead of eagerly loading.

### **Through Unclosed Resources**

* Creating new connection or open a stream, the JVM allocates memory for these resources. A few examples of this include database connections, input streams, and session objects.
* Forgetting to close these resources can block the memory, thus keeping them out of the reach of the GC. This can even happen in case of an exception that prevents the program execution from reaching the statement that’s handling the code to close these resources.

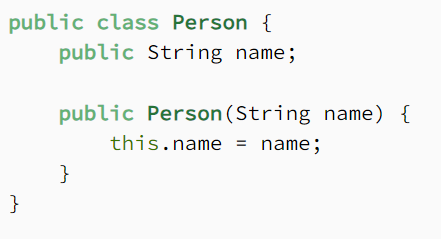
**How to Prevent It?**

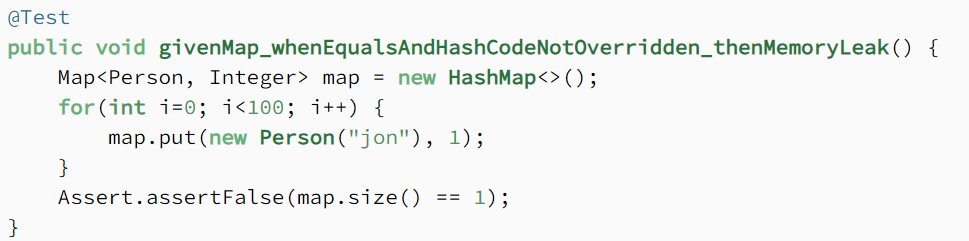
* Always use finally block to close resources.
* The code (even in the finally block) that closes the resources shouldn’t have any exceptions itself.
* Make use of the try-with-resources(Auto closeable ) block.

### **Improper *equals()* and*hashCode()* Implementations**

* When defining new classes, a very common oversight is not writing proper overridden methods for the *equals()* and *hashCode()* methods.
* *HashSet* and *HashMap* use these methods in many operations, and if they’re not overridden correctly, they can become a source for potential memory leak problems.

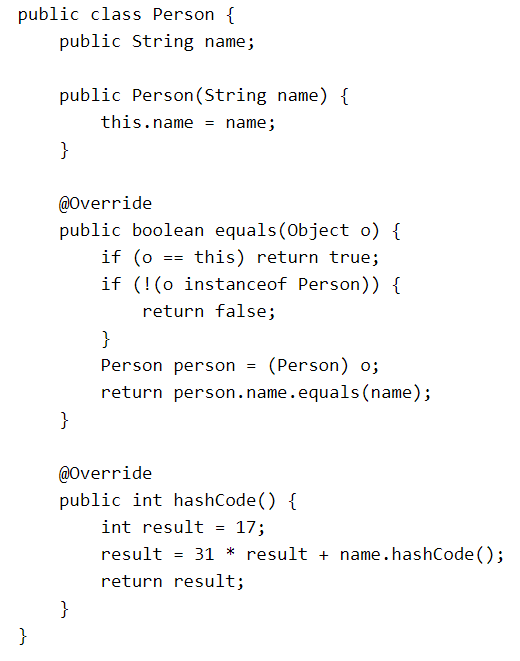
*Eg*





* Here we’re using *Person* as a key. Since *Map* doesn’t allow duplicate keys, the numerous duplicate *Person* objects that we inserted as a key shouldn’t increase the memory.
* But **since we haven’t defined the proper***equals()***method, the duplicate objects pile up and increase the memory**, which is why we see more than one object in the memory. The Heap Memory in VisualVM for this looks like:

**Prevention**



### **Inner Classes That Reference Outer Classes**

* This happens in the case of non-static inner classes (anonymous classes). For initialization, these inner classes always require an instance of the enclosing class.
* Every non-static Inner Class has, by default, an implicit reference to its containing class. If we use this inner class’ object in our application, then **even after our containing class’ object goes out of scope, it won’t be garbage collected**.
* Consider a class that holds the reference to lots of bulky objects and has a non-static inner class. When we create an object of just the inner class, the memory model looks like:

**How to Prevent It?**

* Migrating to the latest version of Java that uses modern Garbage Collectors such as ZGC that uses root references to find unreachable objects. Since the references are found from the root, this will solve the cyclic problem, like the anonymous class holding reference to the container class. More about ZGC [here](https://www.baeldung.com/jvm-zgc-garbage-collector).
* If the inner class doesn’t need access to the containing class members, consider turning it into a static class.

### **Through *finalize()* Methods**

* Use of finalizers is yet another source of potential memory leak issues. Whenever a class’ *finalize()* method is overridden, then **objects of that class aren’t instantly garbage collected.** Instead, the GC queues them for finalization, which occurs at a later point in time.
* Additionally, if the code written in the *finalize()* method isn’t optimal, and if the finalizer queue can’t keep up with the Java garbage collector, then sooner or later our application is destined to meet an *OutOfMemoryError*.
* To demonstrate this, let’s imagine that we have a class for which we’ve overridden the *finalize()* method, and that the method takes a little bit of time to execute. When a large number of objects of this class get garbage collected, it looks like this in VisualVM:

**How to Prevent It?**

* We should always avoid finalizers.

### **Interned *Strings***

* The Java *String* pool went through a major change in Java 7 when it was transferred from PermGen to HeapSpace. However, for applications operating on version 6 and below, we need to be more attentive when working with large *Strings.*
* If we read a massive *String* object, and call *intern()* on that object, it goes to the string pool, which is located in PermGen (permanent memory), and will stay there as long as our application runs. This blocks the memory and creates a major memory leak in our application.

**How to Prevent It?**

* The simplest way to resolve this issue is by upgrading to the latest Java version, as String pool moved to HeapSpace starting with Java version 7.
* If we’re working on large Strings, we can increase the size of the PermGen space to avoid any potential OutOfMemoryErrors:

Eg -XX:MaxPermSize=512m

## Other Strategies for Dealing With Memory Leaks

### **Enable Profiling**

* Java profilers are tools that monitor and diagnose the memory leaks through the application. They analyze what’s going on internally in our application, like how we allocate memory,  like Mission Control, JProfiler, YourKit, Java VisualVM, and the Netbeans Profiler..
* Using profilers, we can compare different approaches and find areas where we can optimally use our resources.

### **Verbose Garbage Collection**

### By enabling verbose garbage collection, we can track the detailed trace of the GC. To enable this, we need to add the following to our JVM configuration:

Eg -verbose:gc

### **Use Reference Objects to Avoid Memory Leaks**

* We can also resort to reference objects in Java that come built-in with the *java.lang.ref* package to deal with memory leaks. Using the *java.lang.ref* package, instead of directly referencing objects, we use special references to objects that allow them to be easily garbage collected.
* Reference queues make us aware of the actions the Garbage Collector performs

### **Eclipse Memory Leak Warnings**

* For projects on JDK 1.5 and above, Eclipse shows warnings and errors whenever it encounters obvious cases of memory leaks. So when developing in Eclipse, we can regularly visit the “Problems” tab and be more vigilant about memory leak warnings (if any):

### **Code Reviews**