Creating a memory leak with Java [closed]

asked 4 years ago

viewed 312230 times

active 3 months ago

1561

I just had an interview, and I was asked to create a memory leak with Java. Needless to say I felt pretty dumb having no clue on how to even start creating one.

What would an example be?

java memory-leaks

1069 share improve this question

edited Aug 28 at 15:51

community wiki
12 revs, 10 users 48%
Mat Banik

closed as too broad by Martijn Pieters ♦ Apr 25 at 16:58

There are either too many possible answers, or good answers would be too long for this format. Please add details to narrow the answer set or to isolate an issue that can be answered in a few paragraphs.

If this question can be reworded to fit the rules in the help center, please edit the question.

115 I would tell them that Java uses a garbage collector, and ask them to be a bit more specific about their definition of "memory leak", explaining that--barring JVM bugs--Java can't leak memory in *quite* the same

way C/C++ can. You have to have a reference to the object *somewhere*. – Darien Jul 7 '11 at 19:00

161 I find it funny that on most answers people are looking for those edge cases and tricks and seem to be completely missing the point (IMO). They could just show code that keep useless references to objects that will never use again, and at the same time never drop those references; one may say those cases are not "true" memory leaks because there are still references to those objects around, but if the program never use those references again and also never drop them, it is completely equivalent to (and as bad as) a "true memory leak". – ehabkost Jul 22 '11 at 6:14

14 Honestly I can't believe the similar question I asked about "Go" got downvoted to -1. Here:stackoverflow.com/questions/4400311/… Basically the memory leaks I was talking about are the ones who got +200 upvotes to the OP and yet I got attacked and insulted for asking if "Go" had the same issue. Somehow I'm not sure that all wiki-thing is working that great. – SyntaxT3rr0r Jul 22 '11 at 11:37

53 @SyntaxT3rr0r - darien's answer is not fanboyism. he explicitly admitted that certain JVMs can have bugs that mean memory gets leaked. this is different than the language spec itself allowing for memory leaks. – Peter Recore Jul 22 '11 at 15:24

8 @ehabkost: No, they are not equivalent. **(1)** You possess the ability to reclaim the memory, whereas in a "true leak" your C/C++ program forgets the range that was allocated, there's no safe way to recover. **(2)**You can very easily detect the problem with profiling, because you can see what objects the "bloat" involves. **(3)** A "true leak" is an unequivocal error, while a program that keeps lots of objects around until terminated *could* be a deliberate part of how it is meant to work. – Darien Jul 22 '11 at 18:37

show **15** more comments

47 Answers

active oldest votes

1 2 next

1181 +50

Here's a good way to create a true memory leak (objects inaccessible by running code but still stored in memory) in pure Java:

1. The application creates a long-running thread (or use a thread pool to leak even faster).
2. The thread loads a class via an (optionally custom) ClassLoader.
3. The class allocates a large chunk of memory (e.g. `new byte[1000000]`), stores a strong reference to it in a static field, and then stores a reference to itself in a ThreadLocal. Allocating the extra memory is optional (leaking the Class instance is enough), but it will make the leak work that much faster.
4. The thread clears all references to the custom class or the ClassLoader it was loaded from.
5. Repeat.

This works because the ThreadLocal keeps a reference to the object, which keeps a reference to its

Linked

11 Easiest way to cause memory leak in Java?

2 Are memory leaks totally absent in Java applications?

0 Create a memory leak given an interface to conform to

2 Is memory leaks possible in java?

74 How do you crash a JVM?

37 What are the roots?

28 Is Go subject to the same subtle memory-leaks that Java is?

14 Can there be memory leak in Java

18 Java vs C++ Qt : what choice for a gentle deployment?

Class, which in turn keeps a reference to its ClassLoader. The ClassLoader, in turn, keeps a reference to all the Classes it has loaded. It gets worse because in many JVM implementations Classes and ClassLoaders are allocated straight into permgen and are never GC'd at all.

A variation on this pattern is why application containers (like Tomcat) can leak memory like a sieve if you frequently redeploy applications that happen to use ThreadLocals in any way. (Since the application container uses Threads as described, and each time you redeploy the application a new ClassLoader is used.)

Update: Since lots of people keep asking for it, here's some example code that shows this behavior in action.

share improve this answer edited Apr 10 at 19:21 community wiki 5 revs, 2 users 72% Daniel Pryden

85 +1 ClassLoader leaks are some of the most commonly painful memory leaks in the JEE world, often caused by 3rd party libs that transform data (BeanUtils, XML/JSON codecs). This can happen when the lib is loaded outside your application's root classloader but holds references to your classes (eg. by caching). When you undeploy/redeploy your app the JVM is unable to garbage collect the app's classloader (and therefore all classes loaded by it), so with repeat deploys the app server eventually borks. If lucky you get a clue with ClassCastException z.x.y.Abc cannot be cast to z.x.y.Abc – earcam Jun 27 '11 at 16:55

4 tomcat uses tricks and nils ALL static variables in ALL loaded classes, tomcat has a lot of dataraces and bad coding though (need to get some time and submit fixes), plus the all mind-boggling ConcurrentLinkedQueue as cache for internal (small) objects, so small that even the ConcurrentLinkedQueue.Node takes more memory. – bestsss Jun 30 '11 at 19:49

26 +1: Classloader leaks are a nightmare. I spent weeks trying to figure them out. The sad thing is, as what @earcam has said, they are mostly caused by 3rd party libs and also most profilers can't detect these leaks. There's a good and clear explanation on this blog about Classloader leaks.blogs.oracle.com/fkieviet/entry/… – Adrian M Jul 8 '11 at 9:08

3 @Nicolas: Are you sure? JRockit does GC Class objects by default, and HotSpot doesn't, but AFAIK JRockit still can't GC a Class or ClassLoader that is referenced by a ThreadLocal. – Daniel Pryden Jul 11 '11 at 17:01

2 Tomcat will try to detect these leaks for you, and warn about them:wiki.apache.org/tomcat/MemoryLeakProtection. The most recent version will sometimes even fix the leak for you. – Matthijs Bierman Mar 2 '12 at 10:33

show **13** more comments

12 Garbage collection of String literals

see more linked questions…

Related

69 How to find a Java Memory Leak

170 Are memory leaks ever ok?

42 Memory leak traps in the Java Standard API

8 Java String.split memory leak?

3 Basics about Java memory leak

1 Java and Memory Leaks

4 Memory leak in java ImageIO.read()

0 Create a memory leak in Java 8

1 How to create memory leaks on Java Web?

1 Java memory leak multithreading

Hot Network Questions

How should I interpret the present perfect in "In no society has nonmarital childbirth been the cultural norm"?

What music notation software allows you to code the notation?

"Most everyone" versus "mostly everyone"?

How to correctly discard old backups?

The Complement Cat

Static field holding object reference [esp final field]

771

```
class MemorableClass {
    static final ArrayList list = new ArrayList(100);
}
```

+50

Calling `String.intern()` on lengthy String

```
String str=readString(); // read lengthy string any source db,textbox/jsp etc..
// This will place the string in memory pool from which you cant remove
str.intern();
```

(Unclosed) open streams (file , network etc...)

```
try {
    BufferedReader br = new BufferedReader(new FileReader(inputFile));
    ...
    ...
} catch (Exception e) {
    e.printStacktrace();
}
```

Unclosed connections

```
try {
    Connection conn = ConnectionFactory.getConnection();
    ...
    ...
} catch (Exception e) {
    e.printStacktrace();
}
```

Areas that are unreachable from JVM's garbage collector, such as memory allocated through native methods

In web applications, some objects are stored in application scope until the application is explicitly stopped or removed.

Disposing of old written notes

How to handle no-call-no-show issues?

Is it bad if my mom comes to a conference with me?

Why do I go through so many philips bits?

How does mhchem print its superscripts and subscripts?

Combinatorial argument for an identity

How do Americans respond when asked for their names?

Sum of orders per customer ID

Chaining Programs

Everything Joe says is true. How can he most help humanity?

Why does having a soul make Angel non-evil compared to evil humans?

What could be the reason for enmity between two races?

How do i `\tag` a subequations environment as a whole?

Cats go Meow, Cows go Moo

Why is the moon hiding from me?

Why are psionics controversial?

Can I enter and exit a country immediately?

How can I work out my girlfriend's ring size, without asking her or using a ring?

How do I create a cut-out hexagon shape?

```
getServletContext().setAttribute("SOME_MAP", map);
```

Incorrect or inappropriate JVM options, such as the `noclassgc` option on IBM JDK that prevents unused class garbage collection

See IBM jdk settings.

share improve this answer edited Jun 30 at 20:32 community wiki 8 revs, 7 users 76% Prashant Bhate

115 I'd disagree that context and session attributes are "leaks." They're just long-lived variables. And the static final field is more or less just a constant. Maybe large constants should be avoided, but I don't think it's fair to call it a memory leak. – Ian McLaird Jul 13 '11 at 4:08

45 *(Unclosed) open streams (file , network etc...)*, doesn't leak for real, during finalization (which will be after the next GC cycle) close() is going to be scheduled (`close()` is usually not invoked in the finalizer thread since might be a blocking operation). It's a bad practice not to close, but it doesn't cause a leak. Unclosed java.sql.Connection is the same. – bestsss Jul 17 '11 at 18:38

15 In most sane JVMs, it appears as though the String class only has a weak reference on its `intern` hashtable contents. As such, it *is* garbage collected properly and not a leak. (but IANAJP)mindprod.com/jgloss/interned.html#GC – Matt B. Jul 22 '11 at 1:32

14 How Static field holding object reference [esp final field] is a memory leak ?? – Kanagavelu Sugumar Sep 10 '12 at 19:52

2 Reference loops used to be leaks as well. That is where you have instance A referencing instance B, which references instance C, which references instance A, but nothing else references any of those. (The loop may need to be bigger) – Graham Feb 10 '13 at 11:58

show **12** more comments

272 A simple thing to do is to use a HashSet with an incorrect (or non-existent) `hashCode()` or `equals()`, and then keep adding "duplicates". Instead of ignoring duplicates as it should, the set will only ever grow and you won't be able to remove them.

If you want these bad keys/elements to hang around you can use a static field like

+50

```
class BadKey {
   // no hashCode or equals();
   public final String key;
   public BadKey(String key) { this.key = key; }
}

Map map = System.getProperties();
map.put(new BadKey("key"), "value"); // Memory leak even if your threads die.
```

share improve this answer edited Jun 24 '11 at 21:50 community wiki 3 revs, 3 users 87% Peter Lawrey

20 Actually, you can remove the elements from a HashSet even if the element class gets hashCode and equals wrong; just get an iterator for the set and use its remove method, as the iterator actually operates on the underlying entries themselves and not the elements. (Note that an unimplemented hashCode/equals is*not* enough to trigger a leak; the defaults implement simple object identity and so you can get the elements and remove them normally.) – Donal Fellows Jun 25 '11 at 17:23

6 @Donal what I'm trying to say, I guess, is I disagree with your definition of a memory leak. I would consider (to continue the analogy) your iterator-removal technique to be a drip-pan under a leak; the leak still exists regardless of the drip pan. – corsiKlause Ho Ho Ho Jun 26 '11 at 20:24

52 I agree, this is **not** a memory "leak", because you can just remove references to the hashset and wait for the GC to kick in, and presto! the memory goes back. – Mehrdad Jul 2 '11 at 4:30

5 @SyntaxT3rr0r, I interpreted your question as asking if there is anything in the language which naturally leads to memory leaks. The answer is NO. This questions asks if it is possible to contrive a situation to create something like a memory leak. None of these examples are memory leak in the manner that a C/C++ programmer would understand it. – Peter Lawrey Jul 22 '11 at 12:32

2 @Peter Lawrey: also, what do you think about this: *"There's not anything in the C language which naturally leaks to memory leak if you don't forget to manually free the memory you allocated"*. How would that be for intellectual dishonesty? Anyway, I'm tired: you can have the last word. – SyntaxT3rr0r Jul 22 '11 at 12:56

show **12** more comments

159 Below there will be non-obvious case where java leaks, besides the standard case of forgotten listeners, static references, bogus/modifiable keys in hashmaps, or just threads stuck w/o any chance to end their life-cycle.

- `File.deleteOnExit()` - always leaks the string, ~~if the string is a substring the leak is even worse (the underlying char[] is also leaked)~~ - *in Java7 substring also copies the* `char[]` *, so the later doesn't apply*; @Daniel, no needs for votes, though.

I'll concentrate on Threads to show the danger of unmanaged threads mostly, don't wish to even touch swing.

- `Runtime.addShutdownHook` and not remove... and then even w/ removeShutdownHook due to a bug in ThreadGroup class regarding unstarted Threads it may not get collected, effectively leak the ThreadGroup. JGroup has the leak in GossipRouter.
- Creating but not starting a `Thread` goes into the same category as above.
- Creating a thread inherits the `ContextClassLoader` and `AccessControlContext`, plus the `ThreadGroup` and any `InheritedThreadLocal`, all those references are potential leaks, along w/ the entire classes loaded by the classloader and all static references, and ja-ja. The effect is especially visible w/ the entire j.u.c.Executor framework that features a super simple `ThreadFactory` interface, yet most developers have no clue of the lurking danger. Also a lot of libraries do start threads upon request (edit: way too many industry popular libraries)
- `ThreadLocal` caches, those are evil in many cases. I am sure everyone has seen quite a bit of simple caches based on ThreadLocal, well the bad news: if the thread keeps going more than expected life the context ClassLoader, it is a pure nice little leak. Do not use ThreadLocal caches unless really needed.
- Calling `ThreadGroup.destroy()` when the ThreadGroup has no Threads itself but still keeps child ThreadGroups. A bad leak that will prevent the ThreadGroup to remove from its parent but all the children become un-enumerateable.
- Using WeakHashMap and the value (in)directly references the key, this is a hard one to find w/o a heap dump. That applies to all extended `Weak/SoftReference` that might keep a hard reference back to the guarded object.
- Using `java.net.URL` w/ http(s) protocol and loading the resource from(!). This one is special, the `KeepAliveCache` creates a new thread in the system ThreadGroup which leaks the current thread's context classloader. The thread is created upon the first request when no alive thread exists, so either you may get lucky or just leak. *The leak is already fixed in java7 and the code that creates thread properly removes the context classloader.* There are few more cases (~~like ImageFetcher~~, *also fixed*) of creating similar threads.
- Using `InflaterInputStream` passing `new java.util.zip.Inflater()` in the constructor (`PNGImageDecoder` for instance) and not calling `end()` of the inflater. Well, if you pass in the constructor w/ just `new`, no chance... and yes calling `close()` on the stream does not close the inflater if it's manually passed as constructor parameter. This is not a true leak since it'd be released by the finalizer... when it deems it necessary. Till that moment it eats native memory so badly it can cause linux oom_killer to kill the process with impunity. The main issue is that

finalization in java is very unreliable and G1 made it worse till 7.0.2. Moral of the story: release native resources as soon as you can, the finalizer is just too poor.

- Same case w/ `java.util.zip.Deflater` , this one is far worse since Deflater is memory hungry in java, i.e. always uses 15bits (max) and 8memory level (9 is max) allocating several hundreds KB of native mem. Fortunately, `Deflater` is not widely used and to my knowledge JDK contains no misuses. Always call `end()` if you manually create a `Deflater` or `Inflater` . The best part of the last two: *you can't find them via normal profiling tools available.*

(I can add some more time wasters I have encountered upon request)

Good luck and stay safe, leaks are evil!

share improve this answer — edited Apr 27 '13 at 14:39 — community wiki 6 revs, 2 users 98% bestsss

7 `Creating but not starting a Thread...` Yikes, I was badly bitten by this one some centuries ago! (Java 1.3) – leonbloy Jul 9 '11 at 1:54

show **1** more comment

99

The answer depends entirely on what the interviewer thought they were asking.

Is it possible in practice to make Java leak? Of course it is, and there are plenty of examples in the other answers.

But there are multiple meta-questions that may have been being asked?

- Is a theoretically "perfect" Java implementation vulnerable to leaks?
- Does the candidate understand the difference between theory and reality?
- Does the candidate understand how garbage collection works?
- Or how garbage collection is supposed to work in an ideal case?
- Do they know they can call other languages through native interfaces?
- Do they know to leak memory in those other languages?
- Does the candidate even know what memory management is, and what is going on behind the scene in Java?

I'm reading your meta-question as "What's an answer I could have used in this interview situation".

And hence, I'm going to focus on interview skills instead of Java. I believe your more likely to repeat the situation of not knowing the answer to a question in an interview than you are to be in a place of needing to know how to make Java leak. So, hopefully, this will help.

One of the most important skills you can develop for interviewing is learning to actively listen to the questions and working with the interviewer to extract their intent. Not only does this let you answer their question the way they want, but also shows that you have some vital communication skills. And when it comes down to a choice between many equally talented developers, I'll hire the one who listens, thinks, and understands before they respond every time.

share improve this answer edited Jan 27 at 12:52 community wiki 3 revs, 2 users 89% PlayTank

11 Whenever I have asked that question, I am looking for a pretty simple answer - keep growing a queue, no finally close db etc, not odd classloader/thread details, implies they understand what the gc can and cannot do for you. Depends on the job you are interviewing for I guess. – DaveC Jul 3 '11 at 17:59

show **1** more comment

74

Most examples here are "too complex". They are edge case. With theses examples, the programmer made a mystake (like don't redifining equals/hashcode), or has been biten by a corner case of the JVM/JAVA (load of class with static...). I think that's not the type of example an interviewer want or even the most common case.

But there are really simpler case to memory leaks. Garbage collector only free what is no longer referenced. We as Java developper don't care about memory. We allocate it when needed and let it being freed automatically. Fine.

But any long lived application tend to have shared state. It can be anything, statics, singletons... Often non trivial application tend to make complex objects graphs. Just forgetting to set a reference to null or more often forgetting to remove one object from a collection is enough to make a memory leak.

Of course all sort of listeners (like UI listeners), caches, or any long lived shared state tend to produce memory leak if not properluy handled. What shall be understood is that this is not a Java corner case, or a problem with the garbage collector. It a design problem. We design that we add listener to a long lived object, but we don't remove the listener when no longer needed. We cache objects, but we have no strategy to remove them from the cache.

We maybe have a complex graph that store previous state that is needed by computation. But the previous state is itself linked to the state before and so on.

Like we have to close SQL connections or files. We need to set proper referenres to null and remove elements from collection. We shall have proper caching strategies (maximum memory size, number of elements, or timers). All object that allow listener to be notified must provide both a addListener and removeListener method. And when theses notifier are no longer used, they must clear their listener list.

MemoryLeak is indeed trully possible and is perfectly predictible. No need for special language features or corner case. Memory leaks are either an indicator that something is maybe missing or even of design problems.

share improve this answer answered Jul 1 '11 at 7:54 community wiki Nicolas Bousquet

5 I find it funny that on other answers people are looking for those edge cases and tricks and seem to be completely missing the point. They could just show code that keep useless references to objects that will never use again, and never remove those references; one may say those cases are not "true" memory leaks because there are still references to those objects around, but if the program never use those references again and also never drop them, it is completely equivalent to (and as bad as) a "true memory leak". – ehabkost Jul 22 '11 at 6:12

show **3** more comments

73

The following is a pretty pointless example, if you do not understand JDBC. Or atleast how JDBC expects a developer to close `Connection`, `Statement` and `ResultSet` instances before discarding them or losing references to them, instead of relying on the implementation of `finalize`.

+50

```
void doWork()
{
   try
   {
       Connection conn = ConnectionFactory.getConnection();
       PreparedStatement stmt = conn.preparedStatement("some query"); // executes a valid qu
       ResultSet rs = stmt.executeQuery();
       while(rs.hasNext())
       {
          ... process the result set
       }
   }
   catch(SQLException sqlEx)
   {
       log(sqlEx);
   }
}
```

The problem with the above is that the `Connection` object is not closed, and hence the physical connection will remain open, until the garbage collector comes around and sees that it is unreachable. GC will invoke the `finalize` method, but there are JDBC drivers that do not implement the `finalize`, at least not in the same way that `Connection.close` is implemented. The resulting behavior is that while memory will be reclaimed due to unreachable objects being collected, resources (including memory) associated with the `Connection` object might simply not be reclaimed.

In such an event where the `Connection`'s `finalize` method does not clean up everything, one might actually find that the physical connection to the database server will last several garbage collection cycles, until the database server eventually figures out that the connection is not alive (if it does), and should be closed.

Even if the JDBC driver were to implement `finalize`, it is possible for exceptions to be thrown during finalization. The resulting behavior is that any memory associated with the now "dormant" object will not be reclaimed, as `finalize` is guaranteed to be invoked only once.

The above scenario of encountering exceptions during object finalization is related to another other scenario that could possibly lead to a memory leak - Object resurrection. Object resurrection is often done intentionally by creating a strong reference to the object from being finalized, from another object. When object resurrection is misused it will lead to a memory leak in combination with other sources of memory leaks.

There are plenty more examples that you can conjure up - like

- managing a `List` instance where you are only adding to the list and not deleting from it (although

you should be getting rid of elements you no longer need), or

- opening `Socket` s or `File` s but not closing them when they are no longer needed (similar to the above example involving the `Connection` class).
- not unloading Singletons when bringing down a Java EE application. Apparently, the Classloader that loaded the Singleton class will retain a reference to the class, and hence the singleton instance will never be collected. When a new instance of the application is deployed, a new class loader is usually created, and the former class loader will continue to exist due to the singleton.

share improve this answer edited Jul 23 '11 at 10:44 community wiki 2 revs Vineet Reynolds

58 You will reach maximum open connection limit before you hit memory limits usually. Don't ask me why I know... – Hardwareguy Jul 21 '11 at 16:23

show **4** more comments

62

Probably one of the simplest examples of a potential memory leak, and how to avoid it, is the implementation of ArrayList.remove(int):

```
public E remove(int index) {
    RangeCheck(index);

    modCount++;
    E oldValue = (E) elementData[index];

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index + 1, elementData, index,
                         numMoved);
    elementData[--size] = null; // (!) Let gc do its work

    return oldValue;
}
```

If you were implementing it yourself, would you have thought to clear the array element that is no longer used (`elementData[--size] = null`)? That reference might keep a huge object alive ...

share improve this answer edited Sep 1 at 4:50 community wiki 3 revs, 3 users 92% meriton

3 'elementData[--size] = null;' does the clearing, I think... – maniek Jul 21 '11 at 18:13

3 And where is the memory leak here? – rds Jul 22 '11 at 16:26

15 @maniek: I did not mean to imply that this code exhibits a memory leak. I quoted to it to show that sometimes non-obvious code is required to avoid accidental object retention. – meriton Jul 23 '11 at 13:26

1 Joshua Bloch gave this example in Effective Java showing a simple implementation of Stacks. A very good answer. – rents Jun 27 at 2:22

show **1** more comment

45 Any time you keep references around to objects that you no longer need you have a memory leak. See Handling memory leaks in Java programs for examples of how memory leaks manifest themselves in Java and what you can do about it.

share improve this answer answered Jun 24 '11 at 16:18 community wiki Bill the Lizard

9 I don't believe this is a "leak". It's a bug, and it's by design of the program and language. A leak would be an object hanging around *without* any references to it. – Mehrdad Jul 2 '11 at 4:31

19 @Mehrdad: That's only one narrow definition that doesn't fully apply to all languages. I'd argue that *any*memory leak is a bug caused by poor design of the program. – Bill the Lizard Jul 10 '11 at 2:30

7 @Mehrdad: `...then the question of "how do you create a memory leak in X?" becomes meaningless, since it's possible in any language.` I don't see how you're drawing that conclusion. There are *fewer* ways to create a memory leak in Java by any definition. It's definitely still a valid question. – Bill the Lizard Jul 10 '11 at 14:09

5 @31eee384: If your program keeps objects in memory that it can never use, then technically it's leaked memory. The fact that you have bigger problems doesn't really change that. – Bill the Lizard Jul 21 '11 at 18:30

7 @31eee384: If you know for a fact that it won't be, then it can't be. The program, as written, will never access the data. – Bill the Lizard Jul 21 '11 at 19:13

show **14** more comments

I can copy my answer from here: Easiest way to cause memory leak in Java?

25 "A memory leak, in computer science (or leakage, in this context), occurs when a computer program consumes memory but is unable to release it back to the operating system." (Wikipedia)

The easy answer is: You can't. Java does automatic memory management and will free resources that are not needed for you. You can't stop this from happening. It will ALWAYS be able to release the resources. In programs with manual memory management, this is different. You cann get some memory in C using malloc(). To free the memory, you need the pointer that malloc returned and call free() on it. But if you don't have the pointer anymore (overwritten, or lifetime exceeded), then you are unfortunately incapable of freeing this memory and thus you have a memory leak.

All the other answers so far are in my definition not really memory leaks. They all aim at filling the memory with pointless stuff real fast. But at any time you could still dereference the objects you created and thus freeing the memory --> NO LEAK. acconrad's answer comes pretty close though as I have to admit since his solution is effectively to just "crash" the garbage collector by forcing it in an endless loop).

The long answer is: You can get a memory leak by writing a library for Java using the JNI, which can have manual memory management and thus have memory leaks. If you call this library, your java process will leak memory. Or, you can have bugs in the JVM, so that the JVM looses memory. There are probably bugs in the JVM, there may even be some known ones since garbage collection is not that trivial, but then it's still a bug. By design this is not possible. You may be asking for some java code that is effected by such a bug. Sorry I don't know one and it might well not be a bug anymore in the next Java version anyway.

share improve this answer answered Jul 2 '11 at 10:23 community wiki yankee

show **1** more comment

24

You able to make memory leak with **sun.misc.Unsafe** class. In fact this service class is used in different standart classes (for example in **java.nio** classes). **You can't create instance of this class directly**, but you may **use reflection to do that**.

Code doesn't compile in Eclipse IDE - compile it using command `javac` (during compilation you'll get warnings)

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import sun.misc.Unsafe;


public class TestUnsafe {

    public static void main(String[] args) throws Exception{
        Class unsafeClass = Class.forName("sun.misc.Unsafe");
        Field f = unsafeClass.getDeclaredField("theUnsafe");
        f.setAccessible(true);
        Unsafe unsafe = (Unsafe) f.get(null);
        System.out.print("4..3..2..1...");
        try
        {
            for(;;)
                unsafe.allocateMemory(1024*1024);
        } catch(Error e) {
            System.out.println("Boom :)");
            e.printStackTrace();
        }
    }

}
```

share improve this answer edited Jul 11 '11 at 15:22 community wiki 2 revs stemm

2 The allocated memory doesn't belong to Java either. – bestsss Jul 17 '11 at 18:35

show **3** more comments

Here's a simple/sinister one via http://wiki.eclipse.org/Performance_Bloopers#String.substring.28.29.

24

```
public class StringLeaker
{
    private final String muchSmallerString;

    public StringLeaker()
    {
        // Imagine the whole Declaration of Independence here
        String veryLongString = "We hold these truths to be self-evident...";

        // The substring here maintains a reference to the internal char[]
        // representation of the original string.
        this.muchSmallerString = veryLongString.substring(0, 1);
    }
}
```

Because the substring refers to the internal representation of the original, much longer string, the original stays in memory. Thus, as long as you have a StringLeaker in play, you have the whole original string in memory, too, even though you might think you're just holding on to a single-character string.

The way to avoid storing an unwanted reference to the original string is to do something like this:

```
...
this.muchSmallerString = new String(veryLongString.substring(0, 1));
...
```

For added badness, you might also `.intern()` the substring:

```
...
this.muchSmallerString = veryLongString.substring(0, 1).intern();
...
```

Doing so will keep both the original long string and the derived substring in memory even after the StringLeaker instance has been discarded.

share improve this answer — edited Jul 21 '11 at 19:00 — community wiki 2 revs Jon Chambers

3 I wouldn't call that a memory leak, *per se*. When `muchSmallerString` is freeed (because the `StringLeaker` object is destroyed), the long string will be freed as well. What I call memory leak is memory that can never been freeed in this instance of JVM. However, you have shown yourself how to free the memory: `this.muchSmallerString=new String(this.muchSmallerString)` . With a real memory leak, there is nothing you can do. – rds Jul 22 '11 at 16:45

2 @rds, that's a fair point. The non- `intern` case may be more of a "memory surprise" than a "memory leak." `.intern()` ing the substring, though, certainly creates a situation where the reference to the longer string is preserved and cannot be freed. – Jon Chambers Jul 22 '11 at 18:42

5 The method substring() creates a new String in java7 (it is a new behavior) – anstarovoyt Mar 22 '13 at 12:43

add a comment

22

Take any web application running in any servlet container (Tomcat, Jetty, Glassfish, whatever...). Redeploy the app 10 or 20 times in a row (it may be enough to simply touch the WAR in the server's autodeploy directory.

Unless anybody has actually tested this, chances are high that you'll get an OutOfMemoryError after a couple of redeployments, because the application did not take care to clean up after itself. You may even find a bug in your server with this test.

The problem is, the lifetime of the container is longer than the lifetime of your application. You have to make sure that all references the container might have to objects or classes of your application can be garbage collected.

If there is just one reference surviving the undeployment of your web app, the corresponding classloader and by consequence all classes of your web app cannot be garbage collected.

Threads started by your application, ThreadLocal variables, logging appenders are some of the usual suspects to cause classloader leaks.

share improve this answer answered Jul 3 '11 at 22:41 community wiki Harald Wellmann

add a comment

19

A common example of this in GUI code is when creating a widget/component and adding a listener to some static/application scoped object and then not removing the listener when the widget is destroyed. Not only do you get a memory leak but a performance hit as when whatever you are listening too fires events all your old listeners are called too.

share improve this answer edited Jul 22 '11 at 10:57 community wiki 2 revs pauli

1 The android platform gives the exemple of a memory leak created by caching a Bitmap in the static field of a

View. – rds Jul 22 '11 at 16:34

add a comment

Maybe by using external native code through JNI?

17 With pure Java, it is almost impossible.

But that is about a "standard" type of memory leak, when you cannot access the memory anymore, but it is still owned by the application. You can instead keep references to unused objects, or open streams without closing them afterwards.

share improve this answer edited Jun 24 '11 at 21:48 community wiki 2 revs, 2 users 67% Rogach

18 That depends on the definition of "memory leak". If "memory that's held on to, but no longer needed", then it's easy to do in Java. If it's "memory that's allocated but not accessible by the code at all", then it gets slightly harder. – Joachim Sauer Jun 24 '11 at 16:15

5 "With pure java, it is almost impossible." Well, my experience is another especially when it comes to implementing caches by people that are not aware of the pitfalls here. – Fabian Barney Jun 24 '11 at 16:28

3 @Rogach: there are basically +400 upvotes on various answers by people with +10 000 rep showing that in both the cases *Joachim Sauer* commented it's very possible. So your "almost impossible" makes no sense.– SyntaxT3rr0r Jul 22 '11 at 11:51

show **1** more comment

I recently encountered a memory leak situation caused in a way by log4j.

15 Log4j has this mechanism called Nested Diagnostic Context(NDC) which is an instrument to distinguish interleaved log output from different sources. The granularity at which NDC works is threads, so it distinguishes log outputs from different threads separately.

In order to store thread specific tags, log4j's NDC class uses a Hashtable which is keyed by the Thread object itself (as opposed to say the thread id), and thus till the NDC tag stays in memory all the objects that hang off of the thread object also stay in memory. In our web application we use NDC to tag logoutputs with a request id to distinguish logs from a single request separately. The container that associates the NDC tag with a thread, also removes it while returning the response from a request. The problem occurred when during the course of processing a request, a child thread was spawned, something like the following code:

```
pubclic class RequestProcessor {
    private static final Logger logger = Logger.getLogger(RequestProcessor.class);
    public void doSomething()  {
        ....
        final List<String> hugeList = new ArrayList<String>(10000);
        new Thread() {
           public void run() {
               logger.info("Child thread spawned")
               for(String s:hugeList) {
                   ....
               }
           }
        }.start();
    }
}
```

So an NDC context was associated with inline thread that was spawned. The thread object that was the key for this NDC context, is the inline thread which has the hugeList object hanging off of it. Hence even after the thread finished doing what it was doing, the reference to the hugeList was kept alive by the NDC context Hastable, thus causing a memory leak.

share improve this answer edited Mar 7 '14 at 21:29 community wiki 2 revs Puneet

show **3** more comments

---

14 I have had a nice "memory leak" in relation to PermGen and xml-parsing once. The XMl-Parser we used (cant remember which one it was) did a String.intern() on tag names, to make comparison faster. One of our customers had the gread idea to store data values not in xml-attributes or text, but as tagnames, so we had a document like:

```
<data>
   <1>bla</1>
   <2>foo</>
   ...
</data>
```

In fact, they did not use numbers but longer textual id's (around 20chars), which where unique and came in at a rate of 10-15 Million a day. That makes 200MB of rubbish a day, which is never needed again, and never GCed (since it is in permgen). We had permgen set to 512MB, so it took around 2 days for the OOME to arrive...

share improve this answer answered Jun 30 '11 at 8:39 community wiki

Ron

3 Just to nitpick your example code: I think numbers (or strings starting with numbers) are not allowed as element names in XML. – Paŭlo Ebermann Jul 3 '11 at 0:18

1 I guess you are right. This was just for demonstration. – Ron Jul 4 '11 at 11:42

add a comment

13 I thought it was interesting that no one used the internal class examples. If you have an internal class; it inherently maintains a reference to the containing class. Of course it is not technically a memory leak because Java WILL eventually clean it up; but this can cause classes to hang around longer than anticipated.

```
public class Example1 {
  public Example2 getNewExample2() {
    return this.new Example2();
  }
  public class Example2 {
    public Example2() {}
  }
}
```

Now if you call Example1 and get an Example2 discarding Example1, you will inherently still have a link to an Example1 object.

```
public class Referencer {
  public static Example2 GetAnExample2() {
    Example1 ex = new Example1();
    return ex.getNewExample2();
  }

  public static void main(String[] args) {
    Example2 ex = Referencer.GetAnExample2();
    // As long as ex is reachable; Example1 will always remain in memory.
  }
}
```

I've also heard a rumor that if you have a variable that exists for longer than a specific amount of time; Java assumes that it will always exist and will actually never try to clean it up if cannot be reached in code anymore. But that is completely unverified.

share improve this answer answered Jul 9 '11 at 1:23 community wiki

Suroot

1 inner classes are rarely an issue. They are a straightforward case and very easy to detect. The rumor is just a rumor too. – bestsss Jul 9 '11 at 6:54

show **1** more comment

12

Create a static Map and keep adding hard references to it. Those will never be GC'd.

```
public class Leaker {
    private static final Map<String, Object> CACHE = new HashMap<String, Object>();

    // Keep adding until failure.
    public static void addToCache(String key, Object value) { Leaker.CACHE.put(key, value);
}
```

share improve this answer edited Jul 21 '11 at 17:32 community wiki 2 revs duffymo

55 How is that a leak? It's doing exactly what you're asking it to do. If that's a leak, creating and storing objects anywhere is a leak. – Falmarri Jul 21 '11 at 19:10

1 I agree with @Falmarri. I don't see a leak there, you are just creating objects. You could certainly 'reclaim' the memory that you just allocated with another method called 'removeFromCache'. A leak is when you can't reclaim the memory. – Kyle Jul 17 '12 at 19:08

2 My point is that somebody who keeps creating objects, perhaps putting them into a cache, could end up with an OOM error if they aren't careful. – duffymo Jul 17 '12 at 20:07

6 @duffymo: But that's not really what the question was asking. It has nothing to do with simply using up all your memory. – Falmarri Jul 18 '12 at 22:10

show **2** more comments

9

As a lot of people have suggested, Resource Leaks are fairly easy to cause - like the JDBC examples. Actual Memory leaks are a bit harder - especially if you aren't relying on broken bits of the JVM to do it for you...

The ideas of creating objects that have a very large footprint and then not being able to access them aren't real memory leaks either. If nothing can access it then it will be garbage collected, and if

something can access it then it's not a leak...

One way that *used* to work though - and I don't know if it still does - is to have a three-deep circular chain. As in Object A has a reference to Object B, Object B has a reference to Object C and Object C has a reference to Object A. The GC was clever enough to know that a two deep chain - as in A <--> B - can safely be collected if A and B aren't accessible by anything else, but couldn't handle the three-way chain...

share improve this answer answered Jul 21 '11 at 17:55 community wiki Graham

4 Hasn't been the case for some time now. Modern GCs know how to handle circular references. – assyliasJun 20 '13 at 23:01

add a comment

9

I came across a more subtle kind of resource leak recently. We open resources via class loader's getResourceAsStream and it happened that the input stream handles were not closed.

Uhm, you might say, what an idiot.

Well, what makes this interesting is: this way, you can leak heap memory of the underlying process, rather than from JVM's heap.

All you need is a jar file with a file inside which will be referenced from Java code. The bigger the jar file, the quicker memory gets allocated.

You can easily create such a jar with the following class:

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.zip.ZipEntry;
import java.util.zip.ZipOutputStream;

public class BigJarCreator {
    public static void main(String[] args) throws IOException {
        ZipOutputStream zos = new ZipOutputStream(new FileOutputStream(new File("big.jar")))
        zos.putNextEntry(new ZipEntry("resource.txt"));
        zos.write("not too much in here".getBytes());
        zos.closeEntry();
        zos.putNextEntry(new ZipEntry("largeFile.out"));
        for (int i=0 ; i<10000000 ; i++) {
            zos.write((int) (Math.round(Math.random()*100)+20));
        }
        zos.closeEntry();
        zos.close();
    }
}
```

Just paste into a file named BigJarCreator.java, compile and run it from command line:

```
javac BigJarCreator.java
java -cp . BigJarCreator
```

Et voilà: you find a jar archive in your current working directory with two files inside.

Let's create a second class:

```
public class MemLeak {
    public static void main(String[] args) throws InterruptedException {
        int ITERATIONS=100000;
        for (int i=0 ; i<ITERATIONS ; i++) {
            MemLeak.class.getClassLoader().getResourceAsStream("resource.txt");
        }
        System.out.println("finished creation of streams, now waiting to be killed");

        Thread.sleep(Long.MAX_VALUE);
    }

}
```

This class basically does nothing, but create unreferenced InputStream objects. Those objects will be garbage collected immediately and thus, do not contribute to heap size. It is important for our example

to load an existing resource from a jar file, and size does matter here!

If you're doubtful, try to compile and start the class above, but make sure to chose a decent heap size (2 MB):

```
javac MemLeak.java
java -Xmx2m -classpath .:big.jar MemLeak
```

You will not encounter an OOM error here, as no references are kept, the application will keep running no matter how large you chose ITERATIONS in the above example. The memory consumption of your process (visible in top (RES/RSS) or process explorer) grows unless the application gets to the wait command. In the setup above, it will allocate around 150 MB in memory.

If you want the application to play safe, close the input stream right where it's created:

```
MemLeak.class.getClassLoader().getResourceAsStream("resource.txt").close();
```

and your process will not exceed 35 MB, independent of the iteration count.

Quite simple and surprising.

share improve this answer — answered Sep 12 '12 at 20:07 — community wiki Jay

add a comment

8 I don't think anyone has said this yet: you can resurrect an object by overriding the finalize() method such that finalize() stores a reference of this somewhere. The garbage collector will only be called once on the object so after that the object will never destroyed.

share improve this answer — answered Jul 3 '11 at 17:36 — community wiki Ben

10 This is untrue. `finalize()` will not be called but the object will be collected once there won't be more references. Garbage collector is not 'called' either. – bestsss Jul 5 '11 at 19:38

1 This answer is misleading, the `finalize()` method can only be called once by the JVM, but this does not mean that it cannot be re-garbage collected if the object is resurrected and then dereferenced again. If there is resource closing code in the `finalize()` method then this code will not get run again, this may cause a memory leak. – Tom Cammann Dec 19 '12 at 11:36

add a comment

8

Everyone always forgets the native code route. Here's a simple formula for a leak:

1. Declare native method.
2. In native method, call `malloc`. Don't call `free`.
3. Call the native method.

Remember, memory allocations in native code come from the JVM heap.

share improve this answer answered Jul 21 '11 at 20:52 community wiki Paul Morie

show **1** more comment

8

You can create a moving memory leak by creating a new instance of a class in that class's finalize method. Bonus points if the finalizer creates multiple instances. Here's a simple program that leaks the entire heap in sometime between a few seconds and a few minutes depending on your heap size:

```
class Leakee {
    public void check() {
        if (depth > 2) {
            Leaker.done();
        }
    }
    private int depth;
    public Leakee(int d) {
        depth = d;
    }
    protected void finalize() {
        new Leakee(depth + 1).check();
        new Leakee(depth + 1).check();
    }
}

public class Leaker {
    private static boolean makeMore = true;
    public static void done() {
        makeMore = false;
    }
    public static void main(String[] args) throws InterruptedException {
        // make a bunch of them until the garbage collector gets active
        while (makeMore) {
            new Leakee(0).check();
        }
        // sit back and watch the finalizers chew through memory
        while (true) {
            Thread.sleep(1000);
            System.out.println("memory=" +
                    Runtime.getRuntime().freeMemory() + " / " +
                    Runtime.getRuntime().totalMemory());
        }
    }
}
```

share improve this answer

answered Jul 22 '11 at 8:05

community wiki
sethobrien

add a comment

8

What's a memory leak:

- It's caused by a **bug** or **bad design.**
- It's a waste of memory.

- **It gets worse over time.**
- **The garbage collector cannot clean it.**

Typical example:

A cache of objects is a good starting point to mess things up.

```
private static final Map<String, Info> myCache = new HashMap<>();

public void getInfo(String key)
{
    // uses cache
    Info info = myCache.get(key);
    if (info != null) return info;

    // if it's not in cache, then fetch it from the database
    info = Database.fetch(key);
    if (info == null) return null;

    // and store it in the cache
    myCache.put(key, info);
    return info;
}
```

Your cache grows and grows. And pretty soon the entire database gets sucked into memory. A better design uses an LRUMap (Only keeps recently used objects in cache).

Sure, you can make things a lot more complicated:

- using **ThreadLocal** constructions.
- adding more **complex reference trees**.
- or leaks caused by **3rd party libraries**.

What often happens:

If this Info object has references to other objects, which again have references to other objects. In a way you could also consider this to be some kind of memory leak, (caused by bad design).

share improve this answer edited Oct 28 '14 at 13:18 community wiki 3 revs bvdb

add a comment

there are many different situations memory will leak. One i encountered, which expose a map that should not be exposed and used in other place.

7

```
public class ServiceFactory {

private Map<String, Service> services;

private static ServiceFactory singleton;

private ServiceFactory() {
    services = new HashMap<String, Service>();
}

public static synchronized ServiceFactory getDefault() {

    if (singleton == null) {
        singleton = new ServiceFactory();
    }
    return singleton;
}

public void addService(String name, Service serv) {
    services.put(name, serv);
}

public void removeService(String name) {
    services.remove(name);
}

public Service getService(String name, Service serv) {
    return services.get(name);
}

// the problematic api, which expose the map.
//and user can do quite a lot of thing from this api.
//for example, create service reference and forget to dispose or set it null
//in all this is a dangerous api, and should not expose
public Map<String, Service> getAllServices() {
    return services;
}
```

share improve this answer

answered Jul 10 '11 at 7:46

community wiki
Ben Xu

add a comment

7

Threads are not collected until they terminate. They serve as roots of garbage collection. They are one of the few objects that won't be reclaimed simply by forgetting about them or clearing references to them.

Consider: the basic pattern to terminate a worker thread is to set some condition variable seen by the thread. The thread can check the variable periodically and use that as a signal to terminate. If the variable is not declared `volatile`, then the change to the variable might not be seen by the thread, so it won't know to terminate. Or imagine if some threads want to update a shared object, but deadlock while trying to lock on it.

If you only have a handful of threads these bugs will probably be obvious because your program will stop working properly. If you have a thread pool that creates more threads as needed, then the obsolete/stuck threads might not be noticed, and will accumulate indefinitely, causing a memory leak. Threads are likely to use other data in your application, so will also prevent anything they directly reference from ever being collected.

As a toy example:

```
static void leakMe(final Object object) {
    new Thread() {
        public void run() {
            Object o = object;
            for (;;) {
                try {
                    sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {}
            }
        }
    }.start();
}
```

Call `System.gc()` all you like, but the object passed to `leakMe` will never die.

(*edited*)

share improve this answer

edited Oct 4 '13 at 4:33

community wiki
2 revs
Boann

1 @Spidey If you would count memory that the process knows about as not being leaked, then all the answers here are wrong, since the process always tracks which pages in its virtual address space are mapped. When the process terminates, the OS cleans up all the leaks by putting the pages back on the free page stack. To take that to the next extreme, one could beat to death any argued leak by pointing out that none of the physical bits in the RAM chips or in the swap space on disk have been physically misplaced or destroyed,

so you can switch the computer off and on again to clean up any leak. – Boann Sep 27 '13 at 18:00

1 @Spidey "I could accept that as a memory leak" Thank you. Earlier, you'd said something couldn't be a leak if it's still referenced and that you couldn't have a leak in Java because it has a garbage collector. >_< "Your answer suggests that any unused allocated object implies a leak". Well, the example is a scenario of an object that the garbage collector can't/won't collect. If it emerges in a program and happens repeatedly it becomes a leak. I'll try to improve my answer with that info. – Boann Oct 4 '13 at 4:30

show **16** more comments

5

I think that a valid example could be using ThreadLocal variables in an environment where threads are pooled.

For instance, using ThreadLocal variables in Servlets to communicate with other web components, having the threads being created by the container and maintaining the idle ones in a pool. ThreadLocal variables, if not correctly cleaned up, will live there until, possibly, the same web component overwrites their values.

Of course, once identified, the problem can be solved easily.

share improve this answer answered Jul 3 '11 at 6:49 community wiki mschonaker

add a comment

5

The interviewer might have be looking for a circular reference solution:

```
public static void main(String[] args) {
    while (true) {
        Element first = new Element();
        first.next = new Element();
        first.next.next = first;
    }
}
```

This is a classic problem with reference counting garbage collectors. You would then politely explain that JVMs use a much more sophisticated algorithm that doesn't have this limitation.

-Wes Tarle

share improve this answer answered Jul 4 '11 at 15:06 community wiki Wesley Tarle

8 *This is a classic problem with reference counting garbage collectors.* Even 15 years ago Java didn't use ref counting. Ref. counting is also slower than GC. – bestsss Jul 5 '11 at 19:36

1 @Esben At each iteration, the previous `first` is not useful and should be garbage collected. In *reference counting* garbage collectors, the object wouldn't be freeed because there is an active reference on it (by itself). The infinite loop is here to desmonstrate the leak: when you run the program, the memory will raise indefinitely. – rds Jul 22 '11 at 16:37

show **1** more comment

5

An example I recently fixed is creating new GC and Image objects, but forgetting to call dispose() method.

GC javadoc snippet:

> Application code must explicitly invoke the GC.dispose() method to release the operating system resources managed by each instance when those instances are no longer required. This is particularly important on Windows95 and Windows98 where the operating system has a limited number of device contexts available.

Image javadoc snippet:

> Application code must explicitly invoke the Image.dispose() method to release the operating system resources managed by each instance when those instances are no longer required.

share improve this answer

answered Jul 21 '11 at 17:41

community wiki
Scott

add a comment

1 2 next

protected by Brad Larson ♦ Apr 15 '13 at 15:25

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?

Not the answer you're looking for? Browse other questions tagged java memory-leaks or ask your own question.