Memory leakage in java

Do you have a Java application that works fine at first but slows down after a while, or does it work normally for a small number of files, but the performance is deteriorating for a large number of files? Maybe you have a memory leak. When fixing memory leaks; If someone asked me: If you knew then what you know now, what would you say to yourself?. Well, I'd say..... Target Audience Although the overall approach described in this article is independent IDE and OS, I have used Linux (Fedora) and Eclipse (plug-in development) in screenshots and instructions. Symptoms of memory leakage work quickly at first, but slows down over time. Works fine with small data sets, serious performance problems with large datasets Increasingly using old generation memory in JVM Out-of-Memory Heap errors in JVM Spontaneous Crash. A common memory leak in Java (who would have thought hehe?) could happen if you forgot to close a resource, or a link to an object is not released. for example, file/text buffers are not closed. (as was the case in my case) hash cards, keeping links alive if equal () and hash code () are not implemented, such as import java.util.Map; MemLeak public class - public final string key; Public MemLeak (String Key) - ;) this.key is the key; map.put (new MemLeak (and quote and key and quote;), and quote and cost;); Details.... Internal classes that refer to external classes may occur. (make them static to avoid). How to fix them? There are two approaches. First, it's an attempt at a quick fix. If this fails, you will have to follow a long path. 1) Fast Fix: Eclipse Memory Leak Warnings (catching some leaks) 2) manually disable and turn on parts of the code and observe the use of your SPM memory using a JVM tool like VisualVM (or Jconsole, or Thermostat). 1) Fast fix: Eclipse Memory Leak Warning/Error. For code matching JDK 1.5 eclipse will throw warnings and errors at you for obvious leaks. More precisely, anything that implements closable (with 1.5) (such as a 1.5 output thread) will throw a warning at you if its link is destroyed, but the object is not closed. However, leaks detection are not always included in eclipse projects. You may have to turn them on first. Go to the project settings and turn them on as forwarded: Now the eclipse will map out a memory leak: However, even with the quirky Eclipse hocus pocus, not all files are closed and leaks detected. Especially when dealing with outdated (up to 1.5) code, you will probably run into leaks because they were written before 'closable' was implemented. Or sometimes the opening/closing file is embedded so deep that the eclipse won't detect them. If you are in this place, you can try Step 2. 2) Manually disable and turn on parts of the code and watch your JVM with a JVM tool like VisualVM. If you've got this far, then you'll have to roll and do some manual labor. You can read all your code and try to understand where the leak is. To help you in the process: I recommend trying to use a tool like VisualVM. (But Thermostat and MAT also work.) Set up VisualVM Download tool Open Terminal, go to.../visualvm_xyz/bin run shell script '../visualvm' (or visualvm.exe on windows). You have to see the main window. If you expand the local and double click on the running app (baby eclipse in my case), you can see its properties. Fixing the VisualVM problem on Fedora (skip this if it works well: For me, initially I couldn't connect to my JVM, I couldn't take a bunch of dumps and profiling didn't work either. Here are a few steps that can help: Make sure you run it as your own user, not sudo ☺. Start VisualVM and then try your luck again. attach VisualVM to your app. Perform an operation that causes sluggish performance. View the Monitor and Memory Pools tab. If you see your memory increasing in the Monitor tab, try clicking the GC (garbage collection) button and see if it reduces memory usage. If not... Then switch to the memory pool tab and inspect Old Gen. (Objects first stick out in Eden, then move through Survivor spaces, old objects move to the 'Old Gen' pool. Repeat until the app is leaking at all. Then, after a few iterations, re-enable parts of the code and check the use of VisualVM memory. When your app starts leaking again, go to the method that caused the memory leak and narrow it down. In the end, you will narrow the problem down to one class, perhaps even one method. Once you're there, carefully confirm that all file buffers are closed and Hashmaps are used properly. Benchmarking code Sometimes it's hard to tell if your shiny new code is better than the old code. In this case, you can sell the performance of the app. Here's some code you can put in the right place to get information about the time and number of garbage collection launches: a long start - (); .. your code.. In the long term - System.currentTimeMillis System.out.println (and quote; Running time: and quote; - Long.toString (end - start)); System.out.println (printGCStats);; public static string printingGCStats () - long totalGarbageCollections No 0; long long long No 0; for (GarbageCollectorMXBean gc : ManagementFactory.getGarbageCollectorMXBeans ()) if (account and gt;) - totalGarbageCollections - count; - Long Time - gc.getCollectionTime (); if (time and 0) Garbage Collections: and quote; - total GarbageCollections - and quote and quote; Garbage Collection Time (ms):and quote; and quote; - garbageCollectionTime; as a note, if you test in the main Eclipse, it is recommended to check in the clean baby Eclipse instead. A: A bunch of landfills I personally haven't used this much, but people are raving about the heap dump. Anytime you can take a pile of landfill and then see how many instances of classes are open and how many places they use. You can double-click on them to see their contents. This is useful if you want to see how many objects your app generates. Hold on, my app has no leaks, but is it still slow? It is possible that you do not have any leaks in the code, but it is still very slow. In this case, you will have to profile your code. Code profiling goes beyond this article, but there's a great YouTube lecture that explains how you can profile with Eclipse using free and paid profilers. Where to go next? At this point, try to spend a day or two actually fixing a memory leak. If you still have a problem, try reading some of these articles: Memory LeaksMemory is one of the most valuable resources developers have at their disposal. Thus, memory efficiency is at the heart of any program you will ever write. The program can be told to be an effective memory when it uses as little memory as possible when in operation at the same time doing what it was designed to do. What is a memory leak? Memory leaks occur when objects are no longer used by your app, but the garbage collector (GC) can't clear them of working memory. This is problematic because these objects pick up memory that might otherwise be used by other parts of your program. Over time, this negates and leads to a deterioration in the performance of the system over time. Garbage collection in JavaJava has grown in popularity largely due to automatic memory management. GC is a program that implicitly cares about memory distribution and transaction distribution. It's a pretty great program and can handle most of the memory leaks that can occur. However, it is not reliable. Memory leaks can still sneak up on an unsuspecting developer, taking away precious resources and, in a pinch, lead to a terrible java.lang.OutOfMemoryError. (It's important to note that OutOfMemoryError isn't necessarily due to memory leaks. large files in memory). Frame prices have reached historic lows in 2019 and are gradually gradually for the last decade or so. Many developers can afford the luxury of never having to deal with insufficient memory, but that doesn't make the problem any less obvious. Android developers are particularly prone to running out of memory because mobile devices have access to far less RAM than their PC counterparts. Most modern phones use LPDDR3 (Low Power Dual Speed 3) RAM compared to the DDR3 and DDR4 components you'll find in most PCs. In other words, while 8GB of RAM is pretty generous for a phone, it's not as powerful as what you get in a PC. In order not to lose focus, more (powerful) RAM does not get rid of the problem completely. The app, riddled with memory leaks, will suffer from serious performance problems as GC uses more and more processors in an attempt to clean unaccounted objects. If the application gets too big, the performance will drop dramatically due to a replacement or will be killed by the system. Memory leaks should be a problem for any

java developer. This article looks at their causes, how to recognize them, and how to deal with them in the app. While there are many subtleties that come with dealing with more limited memory on mobile devices, this article explores memory leaks and how to deal with them in Java SE (ordinary Java most of us are used to). What causes memory leaks? Memory leaks can be caused by a dizzying number of things. The three most common causes are: Misuse of static fields Captured streams Captured Compound) Misused static fields will live in memory as long as the class it belongs to is loaded into JVM - that is, when there are no class instances in JVM. At this stage, the class will be unloaded and the static field will be marked for garbage collection. Catch? Static classes can live in memory mostly forever. Consider the following code: com.memories package; import java.util.ArrayList; import java.util.List; Import java.util.Random; import java.util.logging.Logger; Public class Main - public list list - new ArrayList Public void (Logger.getGlobal); for (int i 0; i zlt; 10000000;i') q list.add (new) random ()));logger.getglobal().info (debug) point 3); logger.getglobal ().info (debug) point) 4); The new ArrayList is a list of new Public i++) { list.add(new Random().nextInt()); Point 3); public static void main(String[] args) { Logger.getGlobal().info(Debug Point 1); new Main().populateList(); Logger.getGlobal().info(Debug Point 4); } catch (InterruptedException e) { As far as programs go, this isn't very 10000000;= i++)= {= list.add(new= random().nextint());= logger.getglobal().info(debug= point= 3);= public= static= void= main(string[]= args)= {= logger.getglobal().info(debug= point= 1);= new= main().populatelist();= logger.getglobal().info(debug= point= 4);= }= catch= (interruptedexception= e)= {= as= far= as= programs= go,= this= isn't= very= ></ 10000000; i++) { list.add(new Random().nextInt()); Logger.getGlobal().info(Debug Point 3); public static void main(String[] args) { Logger.getGlobal().info(Debug Point 1); new Main().populateList(); Logger.getGlobal().info(Debug Point 4); } catch (InterruptedException e) { As far as programs go, this isn't very > пустой populateList () - Logger.getGlobal ().info(Debug Point 2)</Integer> </Integer> </Integer> мы создаем новый класс, который имеет общедоступный ArrayList. Затем мы населяем этот ArrayList миллионом записей. Используя наш удобный инструмент профилирования Java с открытым исходным кодом, VisualVM, мы получаем следующий график после его запуска: VisualVM графикВ 1-секундной отметке размер кучи увеличивается по мере того, как JVM присваивает нашей программе 417 МБ, и в течение дополнительной секунды, необходимой для запуска, JVM очищает все из памяти. Как используемая, так и назначенная память выходят из-под конца, и программа выключается. Давайте сравним это с предыдущим кодом, измененным, чтобы сделать ArrayList статичным: пакет com.memories; импортировать java.util.ArrayList; импортировать java.util.List; импортировать java.util.Random; импортировать java.util.logging.Logger; public class Main - публичный статический список списков - новый ArrayList (); публичная пустота<Integer> <> populateList () - Logger.getGlobal ().info (Точка отладки 2); для (int i q 0; i < 10000000;= i++)= {= list.add(new= random().nextint());= }= logger.getglobal().info(debug= point= 3);= }= public= static= void= main(string[]= args)= {= logger.getglobal().info(debug= point= 1);= new= main().populatelist();= logger.getglobal().info(debug= point= 4);= try= {= system.gc();= thread.sleep(5000);= }= catch= (interruptedexception= e)= {= e.printstacktrace();= }= }= }import= java.util.arraylist;import= java.util.logging.logger;= public= static=> <Integer>list - новый ArrayList <> (); public void populateList () - < 10000000; i++) { list.add(new Random().nextInt()); Logger.getGlobal().info(Debug Point 3); public static void main(String[] args) { Logger.getGlobal().info(Debug Point 1); new Main().populateList(); Logger.getGlobal().info(Debug Point 4); } catch (InterruptedExceptio e) { This time, the heap size increases to ~390MB and after the program is done running (at the point with a slight downward slope) the used memory remains stagnant till the program ends. VisualVM graph -2How to avoid this mistake: Minimize the use of static fields in your application.b) Unclosed streamsIn the context of this article, a memory leak is defined as happening when code holds a reference to an object so that the garbage collector can't do anything about it. By that definition, a closed stream isn't exactly a 'memory leak' (unless you have an unclosed reference to it). However, most OSs will limit how many files (in the case of FileInputStream) an application can have open at once. If such a stream isn't closed, it may take quite some time before the GC realizes those streams need to be closed, so it is a leak, just not a memory leak per se. A more compelling example would be loading a large object using URLConnection (or similar classes). The following code will cause potential issues down the line if we don't close both FileInputStream and ReadableByteChannel. package com.memories; import java.io.*; import java.net.MalformedURLException; import import java.net.URLConnection; import java.nio.channels.Channels; import java.nio.channels.ReadableByteChannel; import 10000000;= i++)= {= list.add(new= random().nextint());= logger.getglobal().info(debug= point= 3);= public= static= void= main(string[]= args)= {= logger.getglobal().info(debug= point= 1);= new= main().populatelist();= logger.getglobal().info(debug= point= 4);= }= catch= (interruptedexception= e)= {= this= time,= the= heap= size= increases= to= ~390mb= and= after= the= program= is= done= running= (at= the= point= with= a= slight= downward= slope)= the= used= memory= remains= stagnant= till= the= program= ends.visualvm= graph= -2how= to= avoid= this= mistake:= minimize= the= use= of= static= fields= in= your= application.b)= unclosed= streamsin= the= context= of= this= article,= a= memory= leak= is= defined= as= happening= when= code= holds= a= reference= to= an= object= so= that= the= garbage= collector= can't= do= anything= about= it.= by= that= definition,= a= closed= stream= isn't= exactly= a= 'memory= leak'= (unless= you= have= an= unclosed= reference= to= it).however ,= most= oss= will= limit= how= many= files= (in= the= case= of= fileinputstream)= an= application= can= have= open= at= once.= if= such= a= stream= isn't= closed,= it= may= take= quite= some= time= before= the= gc= realizes= those= streams= need= to= be= closed,= so= it= is= a= leak,= just= not= a= memory= leak= per= se.a= more= compelling= example= would= be= loading= a= large= object= using= urlconnection= (or= similar= classes).the= following= code= will= cause= potential= issues= down= the= line= if= we= don't= close= both= fileinputstream= and= readablebytechannel.= package= com.memories;= import= java.io.*;= import= java.net.malformedurlexception;= import= java.net.url;= import= java.net.urlconnection;= import= java.nio.channels.channels;= import= java.nio.channels.readablebytechannel;= import=></ 10000000; i++) { list.add(new Random().nextInt()); Logger.getGlobal().info(Debug Point 3); public static void main(String[] args) { Logger.getGlobal().info(Debug Point 1); new Main().populateList(); Logger.getGlobal().info(Debug Point 4); } catch (InterruptedException e) { This time, the heap size increases to ~390MB and after the program is done running (at the point with a slight downward slope) the used memory remains stagnant till the program ends. VisualVM graph -2How to avoid this mistake: Minimize the use of static fields in your application.b) Unclosed streamsIn the context of this article, a memory leak is defined as happening when code holds a reference to an object so that the garbage collector can't do anything about it. By that definition, a closed stream isn't exactly a 'memory leak' (unless you have an unclosed reference to it). However, most OSs will limit how many files (in the case of FileInputStream) an application can have open at once. If such a stream isn't closed, it may take quite some time before the GC realizes those streams need to closed, so it is a leak, just not a memory leak per se. A more compelling example would be loading a large object using URLConnection (or similar classes). The following code will cause potential issues down the line if we don't close both FileInputStream and ReadableByteChannel. package com.memories; import java.io.*; import java.net.MalformedURLException; import java.net.URL; import java.net.URLConnection; import java.nio.channels.Channels; import java.nio.channels.ReadableByteChannel; import > Logger.getGlobal ().info(Debug Pointbug 2)</Integer> </Integer> </Integer> PUBLIC class URLeak - Public static emptiness core (String) throws IOException - URL URL URL -1 URL (; ReadableByteChannel rbc - Channels.newChannel (url.openStream(); FileOutputStreamStream - new FileOutputStream (/); outputStream.getChannel ().transferFrom (rbc, 0, Long.MAX_VALUE); - import java.net.MalformedURLException;import java.net.URLConnection;import java.nio.channels.channels.channels;import java.nio.channels.ReadableByteChannel; import java.nio.charset.Charset; public static void core (String) throws IOException - URL -3 URL (; ReadableByteChannel rbc - Channels.newChannel (url.openStream(); FileOutputStreamStream exit - new FileOutputStream (/); outputStream.getChannel (.transferFrom (rbc, 0, Long.MAX_VALUE); Simple fix is to close threads. //..... outputStream.close(); rbc.close(); (c) Undetected database connections Are a complex problem for debugging. As reflected in the title, I learned this lesson the hard way when implementing some features on my site. Every time there was a movement, nothing would happen. No errors, no exceptions and no glitches, but the server will be time out for each request. Even more strange, once the number of requests has decreased, is just a mysterious error. A small dig will show that at such times, all processes related to the database were in line but never processed. Something was holding the database. In the end, we came across a code that looked like it was littered with everything: the com.memories package; Java.sql.Connection import imports java.sql.PreparedStatement; import java.sql.S'LException; Public Class DBLeak - Public Static Void Core (String) Args) Throws S'LException - Connection Connection - JDBCHelper.getConnection (); Prepared State stm and zero; Try - stmt - connection.prepareStatement (SELECT ...); Catch (S'LException e) - e.printStackTrace - finally - / Release the statement stmt.close (); Notice how the connection never closes. Easy to miss. Imports Java.sql.Connection; import java.sql.PreparedStatement; import java.sql.S'LException; Public static emptiness of the main (String) argy) throws S'LException - Connection Connection - JDBCHelper.getConnection (); Prepared State stm and zero; stmt - connection.prepareStatement (SELECT ...); Catch (S'LException e) / Please note how the connection never closes. Easy to miss. Perhaps the worst example of the same will be: the com.memories package; import java.sql.' public class DBLeak1 - public static emptiness core (String) - attempt - String realName - getRealNameFromDatabase (AFriskyWaterMelon, team90%waterFortheWin); System.out.println - улов (S'LException e) - e.printStackTrace (); публичная статическая строка getRealNameFromDatabase (String username, Струнный пароль) бросает S'LException S'LException Username, password); Statement stmt and con.createStatement(); ResultSet rs - stmt.execute'y (SELECT first_name, last_name from users); Line first - ; The last line is While (rs.next) - the first - rs.getString (first_name); Last-name - rs.getString (last_name); Public Static Void Core (String) - String realName - getRealNameFromDatabase (aFriskyWaterMelon, team90%waterFortheWin); System.out.println Catch (S'LException e) - public static line getRealNameFromDatabase (String username, String password) throws S'LException - Connection Con - DriverManager.getConnection (jdbc:myDriver:devDB, Statement stmt s con.createStatement (); ResultSet rs - stmt.execute'y (SELECT first_name, last_name from users); First name - rs.getString (first_name); lastName - rs.getString (last_name); Return the first name and last name In this case, each resource is leaking. Fortunately, there are several ways to solve this problem: Use ORM: ORMs to take care of opening and closing any resources for you. One of the most popular options is Hibernate.Automatic Resource Management is a long-awaited feature that has finally been introduced in Java 8. It's also known as a try with resources.'Use jOO: JOO' isn't exactly ORM, but it manages all database resources automatically for you. How to detect memory leaksOut profiler is a tool that allows you to monitor various aspects of JVM, including performing flow and collecting garbage. This is useful if you want to compare different approaches and find which one is most effective in functionality like memory distribution. In this tutorial we used VisualVM, but tools such as Flight Control, Netbeans Profiler and JProfiler are all available if VisualVM doesn't fit your fancy. Using heaps of landfillsIf you don't want to learn how to use another new tool, a bunch of landfills can help. The heap dump is an instant snapshot of all objects in JVM's memory anyway. A bunch of landfills lets you see how much space certain objects in JVM take up at any given point. They're useful for knowing how many objects your app generates.NB: We haven't reviewed the final section in great depth, but that's because detecting memory leaks is a whole beast in itself. She needs her own article to cover all the little details that might otherwise blindsided you. Memory summary leaks are a pressing issue for most developers and should not be taken lightly. They are difficult to detect and may be even harder to solve if they occur in production, eventually leading to fatal application failures. However, following best practices, such as writing tests and code reviews and profiling, you can minimize the likelihood of memory loss in Applications. Next part: Hunting down and fixing memory leaks on Java Java Java Java memory leak in java. memory leakage in javascript. memory leak in java solution. memory leak in java detection tool. memory leak in java javatpoint. memory leak in java android. memory leak in java 8. memory leak in java intellij