Tools and Strategies for Tracing Memory Usage and Leaks

Justin Farnsworth, Matt Mozdzen, Matthew Van Soelen, Brian Worts

To Contact for further information on this topic: farnswj1@tcnj.edu, mozdzem1@tcnj.edu, vansoem1@tcnj.edu, and wortsb1@tcnj.edu

Canvas Page: https://tcnj.instructure.com/groups/364604

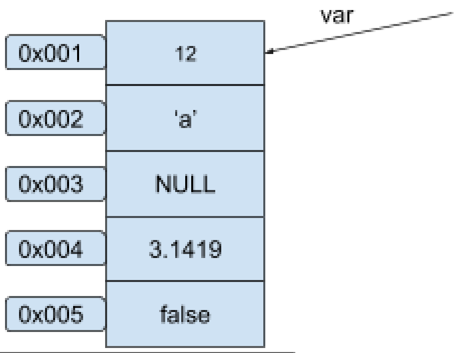
What is a Pointer?

A *pointer* is a reference to a memory location of a variable. In C, you can declare a pointer variable like so:

```
int x = 12;

int *var = &x;

//var would have the value “0x001
```



The structure of heap memory depends on the architecture of the computer. Some heaps may only be able to hold 32 bits in a location, while more modern ones can hold 64 bits.

Pointers in Ruby

In Ruby, variables are pointers to objects. However, Ruby can support the usage of C-type pointers with the class Fiddle::Pointer.

You can declare a pointer as such:

Fiddle::Pointer.new(address) → fiddle_cptr

#declare a c-type pointer

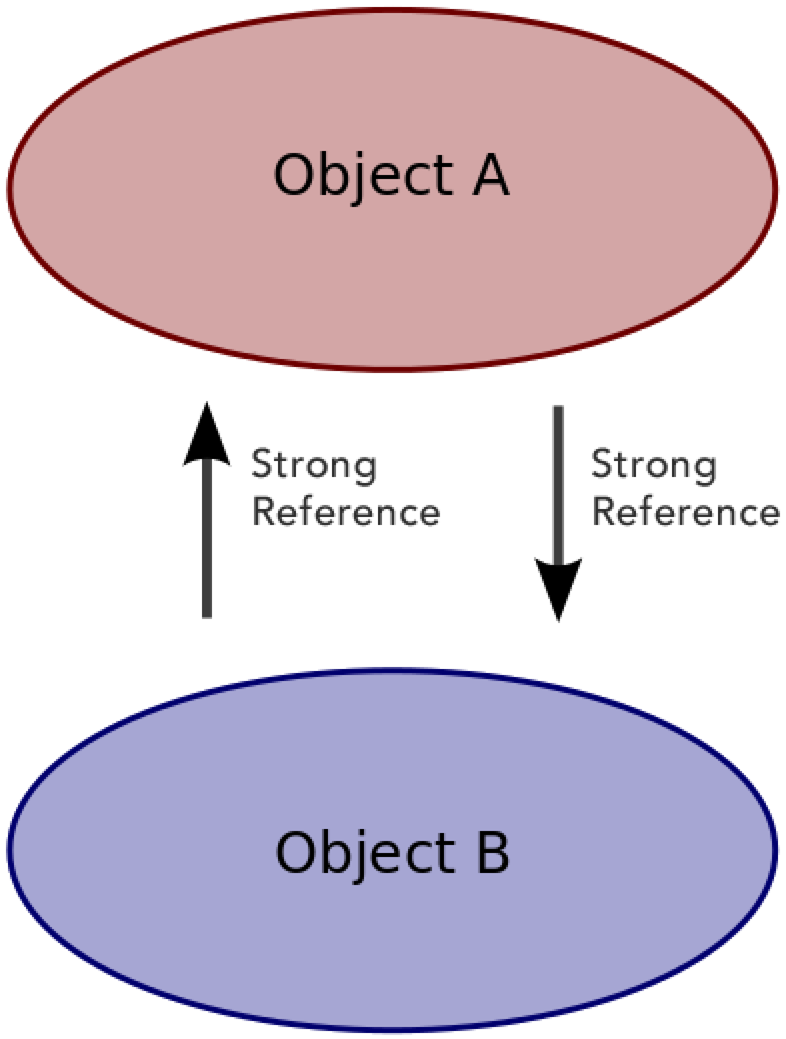
What is Memory Tracing?

Memory tracing is the process of monitoring and recording any changes in a system's memory. This makes it a very effective tool for debugging. Most approaches involve executing additional code every time a memory address is accessed. As a result, memory tracing can increase performance overhead by up to 10 times in some cases, which is why it is usually not enabled by default.

What is a Memory Leak?

A *memory leak* is an event that occurs when memory is allocated for an object/variable but continues to take up that memory after it is no longer needed by a running program, thus resulting in decreased available heap memory for other processes. A very common sign of a memory leak is that the longer an application is running, the more memory gets consumed. If a memory leak occurs, they can negatively impact performance and even cause failure/crashing of the application.

Causes



Memory leaks are caused by poor programing and fall into many categories. In general, they result from a program not properly deallocating blocks of memory while running or even after being closed.

One such reason for a memory leak is a "*strong reference cycle*." This means that objects contain references to each other, thus preventing deallocation of the memory.

Another such reason is that, as a program runs a block of code repeatedly, instead of allocating that space and then reallocating that space for the next iteration, it allocates a new chunk of memory each time.

Solutions for Memory Leaks

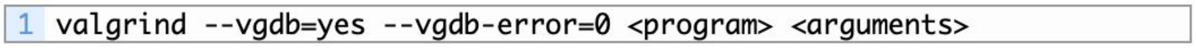
On the user end, one of the simplest ways to resolve problems caused by a memory leak is to close the application causing the leak or, if that does not work, then restarting their computer will. Restarting the computer “clears” the memory of the computer by deallocating used-up space. However, running the program that caused the memory leak again will cause another buildup in the memory and the computer will need to be restarted again. Thus, the only way to stop a memory leak in a program is for the developer to remove it from the code, which is why memory tracing is such an important skill for developers to know.

Commonly Used Tools for Memory Tracing

GDB (GNU Debugger)

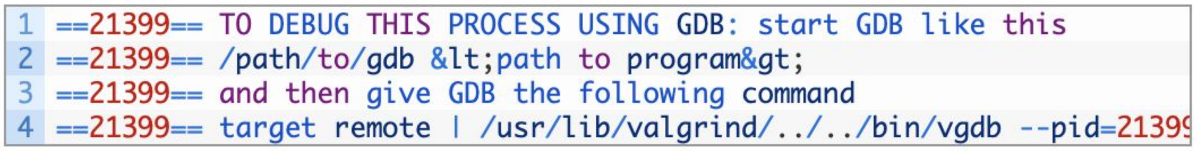
Useful for debugging C and C++ programs, mainly for identifying problems involving memory leaks. To debug memory leaks with GDB, you will need to use the built-in tool Valgrind.

Step 1: Launch Valgrind



```
valgrind --vgdb=yes --vgdb-error=0 <program> <arguments>
```

STEP 2: Follow the printed instructions to run the gdb



```
==21399== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==21399== /path/to/gdb &lt;path to program&gt;
==21399== and then give GDB the following command
==21399== target remote | /usr/lib/valgrind/../../bin/vgdb --pid=2139
```

STEP 3: Launch the gdb for your program

```
gdb <program>
```

STEP 4: Disable Non-Stop Mode

```
gdb> set non-stop off
```

STEP 5: Attach Valgrind to your gdb

```
gdb> target remote | vgdb
```

STEP 6: Check for memory leaks

```
gdb> monitor leak_check
```

Now, run the program and leak check at an occasional interval. Stop it when a leak is found. Restart the program and do this, making the interval smaller each time, until you can find where the leak is coming from. If you have an idea of where the leak is, put a break statement after it in your code.

GDB Manual: https://sourceware.org/gdb/current/onlinedocs/gdb/

Source:
http://www.responsive.se/thomas/2013/09/20/debugging-memory-leaks-with-valgrind-and-gdb/

Garbage Collectors

Some languages, such as JAVA, have a built-in program that automatically manages memory. This makes cleaning up memory more convenient the programmer doesn’t have to manually clean it up.

However, it doesn’t necessarily mean that garbage collectors are faster than manual memory management. If the programmer knows what needs to be cleaned up, then it may be efficient to manually clean it up, as the garbage collector will allocate possibly more time and resources to do so. Some garbage collectors will even pause the program while it cleans up memory.

Task Manager

Although minimal detail is given, the “Task Manager” (on Windows) can be used to monitor RAM usage (along with some other hardware component usage). This is useful when a user is running many applications but does not know which one is slowing down their computer's performance.

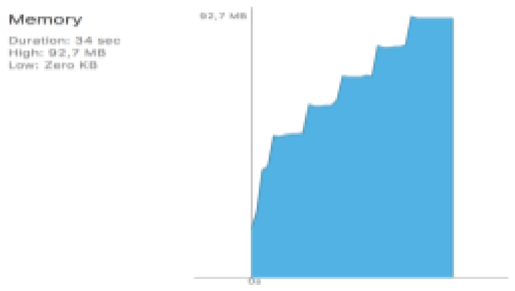
Memory Tracing with XCode

Memory Graph Debugger

This is a tool provided by XCode to help developers find memory leaks for XCode versions 8.0 and up. It visualizes all the objects in memory while the program is running, and then guides the user to where that memory leak is occuring.

In order to visualize the memory use, this tool provides a helpful graph in real time of the memory being used while the program is running. A memory leak can be clearly seen when, for example, the user hits a button over and over again and each time, the memory use of the program increases.

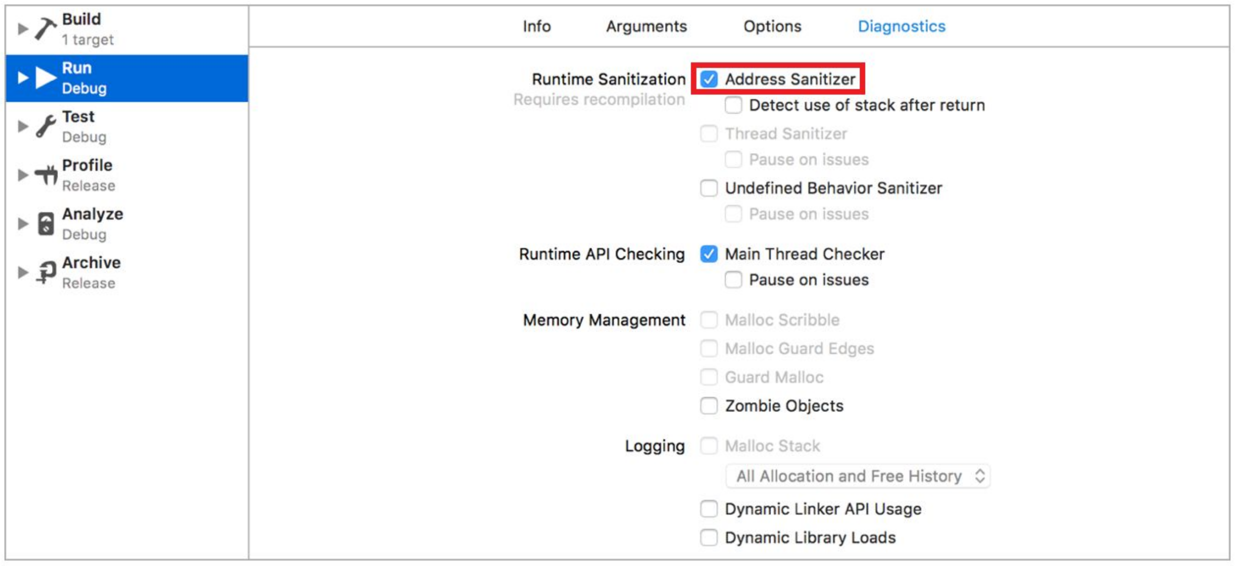
Then, to help find the leak, objects stored in the memory are shown. The user can scan this list and should they find multiple instances of the same object in the heap, it becomes obvious what is causing the leak.



The Memory Graph Debugger tool showing memory usage over time of an application repeatedly running the same function with a memory leak in it.
Source:
https://medium.com/@imas145/using-memory-graph-debugger-in-xcode-9-a50aa9d0243f

Examples of how to use this tool are given in Hands On Examples 1 and 2.

Another useful tool in XCode is the *Address Sanitizer*, which adds instrumentation to your app that enables Xcode to stop your app where memory corruption happens. Address sanitizer finds problems such as accessing deallocated pointers, buffer overflow and underflow of the heap and stack, and other memory issues. To use address sanitizer, enable it in the debug scheme for your target, then run and use the app. Xcode monitors memory use and stops your app on the line of code causing the problem and opens the debugger. Use the debugger to isolate the cause.



Runtime Sanitization
Requires recompilation
- ✓ Address Sanitizer
  - ☐ Detect use of stack after return
- ☐ Thread Sanitizer
  - ☐ Pause on issues
- ☐ Undefined Behavior Sanitizer
  - ☐ Pause on issues

Runtime API Checking
- ✓ Main Thread Checker
  - ☐ Pause on issues

Memory Management
- ☐ Malloc Scribble
- ☐ Malloc Guard Edges
- ☐ Guard Malloc
- ☐ Zombie Objects

Logging
- ☐ Malloc Stack
  - All Allocation and Free History
- ☐ Dynamic Linker API Usage
- ☐ Dynamic Library Loads

Running your code with *Address Sanitizer* checks enabled typically results in CPU slowdown of

2✕ to 5✕, and an increase in memory usage by 2✕ to 3✕. You can improve memory utilization by compiling at the -O(1) optimization level.

For most use cases, the overhead introduced by the Address Sanitizer should be acceptable for daily use during development—in fact, you may not even notice any slowdown.

Further Reading

https://useyourloaf.com/blog/xcode-visual-memory-debugger/

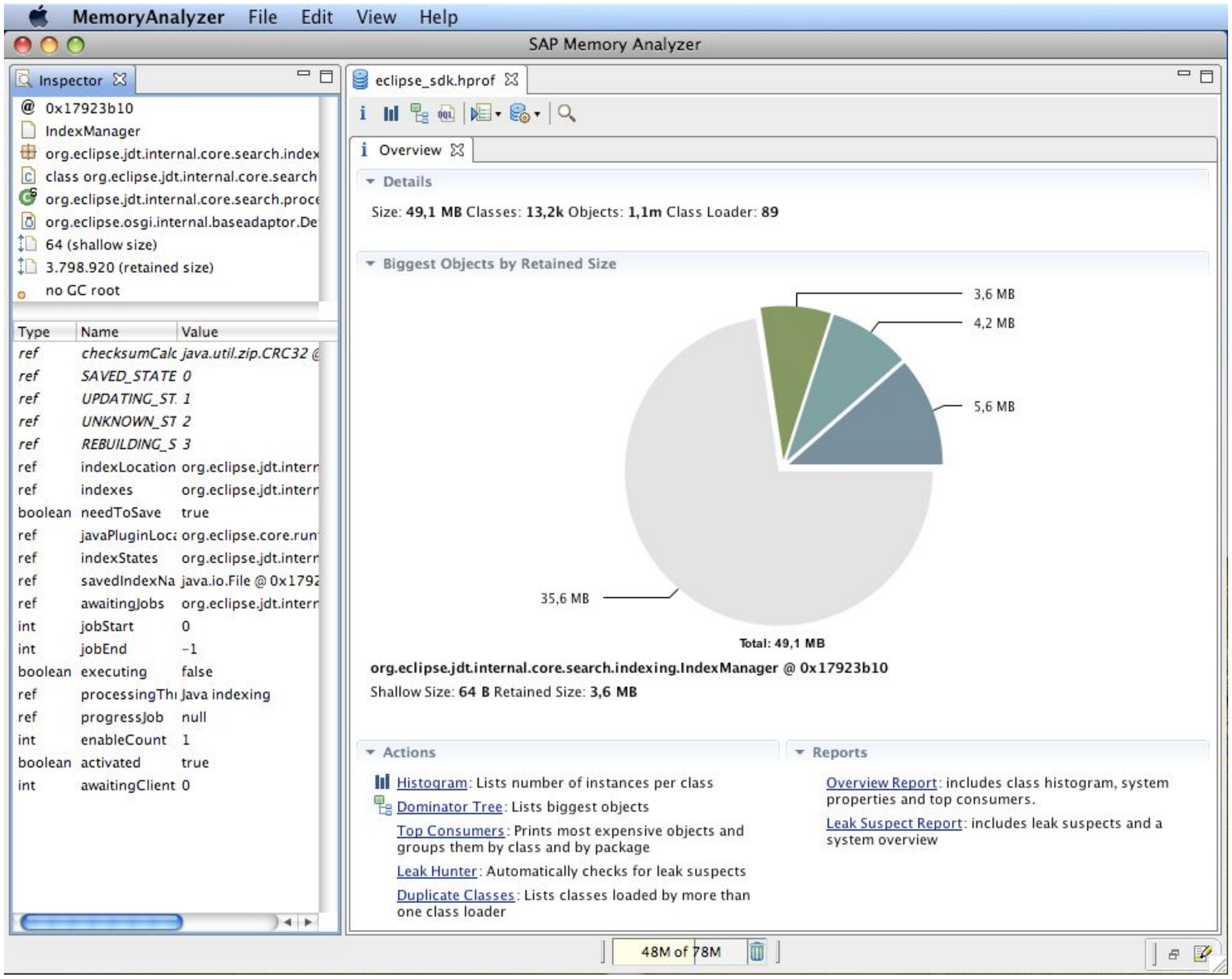
https://medium.com/@imas145/using-memory-graph-debugger-in-xcode-9-a50aa9d0243f

https://agostini.tech/2018/12/09/memory-graph-debugging-in-xcode/
https://www.surekhatech.com/blog/find-memory-leaks-in-ios-apps-with-xcode-instruments

https://developer.apple.com/documentation/code_diagnostics/address_sanitizer

Eclipse Memory Analyzer (MAT)

Eclipse allows you to monitor memory usage and find memory leaks. It is particularly useful for Java heap-dumps. Conveniently, MAT visualizes memory usage across programs through various diagrams, such as pie charts and histograms.



@ 0x17923b10
IndexManager
org.eclipse.jdt.internal.core.search.index
class org.eclipse.jdt.internal.core.search
org.eclipse.jdt.internal.core.search.proce
org.eclipse.osgi.internal.baseadaptor.De
64 (shallow size)
3.798.920 (retained size)
no GC root

Type	Name	Value
ref	*checksumCalc*	*java.util.zip.CRC32 @*
ref	*SAVED_STATE*	*0*
ref	*UPDATING_ST.*	*1*
ref	*UNKNOWN_ST*	*2*
ref	*REBUILDING_S*	*3*
ref	indexLocation	org.eclipse.jdt.interr
ref	indexes	org.eclipse.jdt.interr
boolean	needToSave	true
ref	javaPluginLoc	org.eclipse.core.run
ref	indexStates	org.eclipse.jdt.interr
ref	savedIndexNa	java.io.File @ 0x1792
ref	awaitingJobs	org.eclipse.jdt.interr
int	jobStart	0
int	jobEnd	-1
boolean	executing	false
ref	processingThı	Java indexing
ref	progressJob	null
int	enableCount	1
boolean	activated	true
int	awaitingClient	0

Details

Size: **49,1 MB** Classes: **13,2k** Objects: **1,1m** Class Loader: **89**

Biggest Objects by Retained Size

org.eclipse.jdt.internal.core.search.indexing.IndexManager @ 0x17923b10

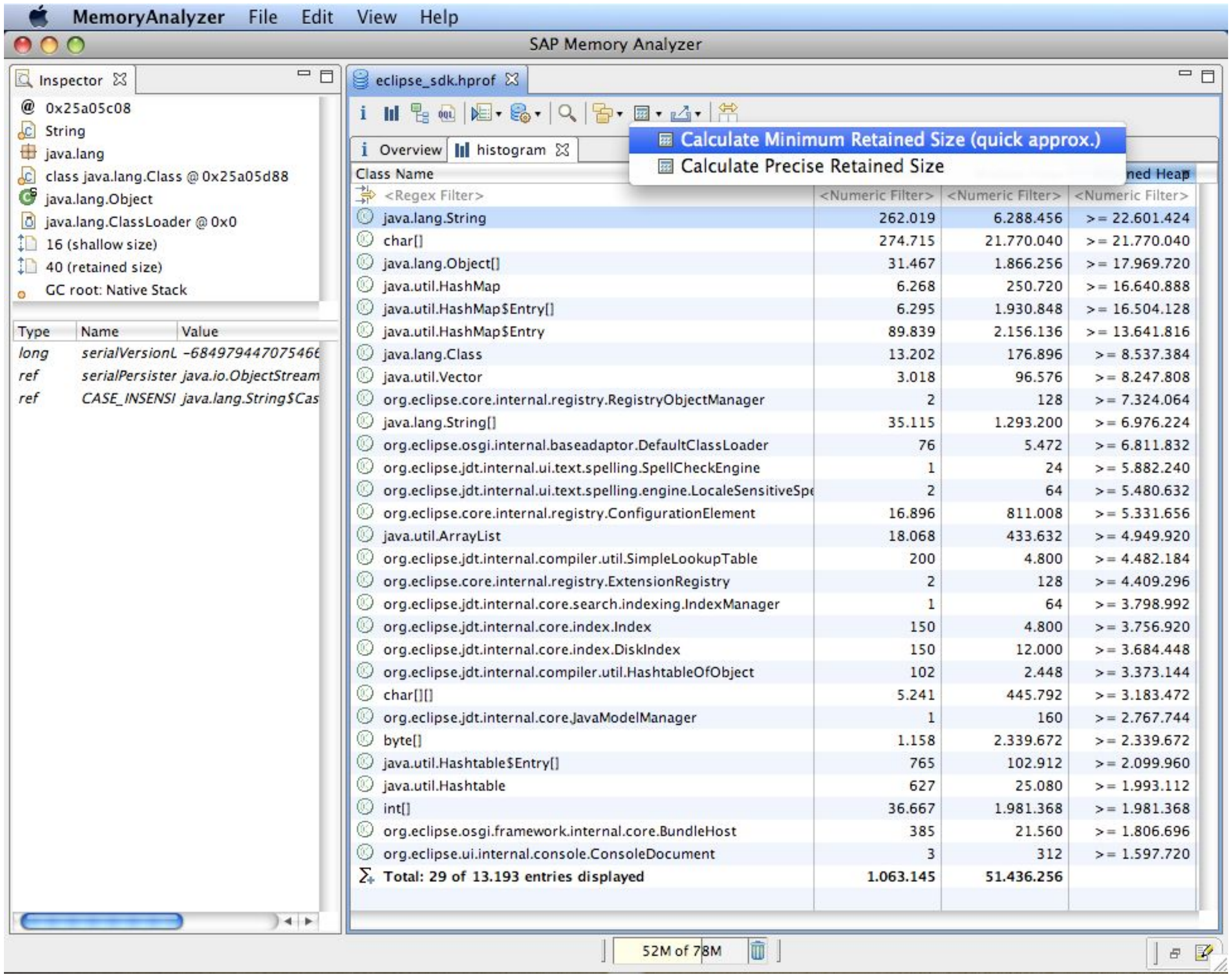
Shallow Size: **64 B** Retained Size: **3,6 MB**

Actions

- Histogram: Lists number of instances per class
- Dominator Tree: Lists biggest objects
- Top Consumers: Prints most expensive objects and groups them by class and by package
- Leak Hunter: Automatically checks for leak suspects
- Duplicate Classes: Lists classes loaded by more than one class loader

Reports

- Overview Report: includes class histogram, system properties and top consumers.
- Leak Suspect Report: includes leak suspects and a system overview


MemoryAnalyzer File Edit View Help
SAP Memory Analyzer
Inspector
eclipse_sdk.hprof
Overview
histogram
Calculate Minimum Retained Size (quick approx.)
Calculate Precise Retained Size
52M of 78M

Inspector
0x25a05c08
String
java.lang
class java.lang.Class @ 0x25a05d88
java.lang.Object
java.lang.ClassLoader @ 0x0
16 (shallow size)
40 (retained size)
GC root: Native Stack

Type	Name	Value
long	*serialVersionL*	*-68497944707546*
ref	*serialPersister*	*java.io.ObjectStream*
ref	*CASE_INSENSI*	*java.lang.String$Cas*

Class Name			ned Heap
<Regex Filter>	<Numeric Filter>	<Numeric Filter>	<Numeric Filter>
java.lang.String	262.019	6.288.456	>= 22.601.424
char[]	274.715	21.770.040	>= 21.770.040
java.lang.Object[]	31.467	1.866.256	>= 17.969.720
java.util.HashMap	6.268	250.720	>= 16.640.888
java.util.HashMap$Entry[]	6.295	1.930.848	>= 16.504.128
java.util.HashMap$Entry	89.839	2.156.136	>= 13.641.816
java.lang.Class	13.202	176.896	>= 8.537.384
java.util.Vector	3.018	96.576	>= 8.247.808
org.eclipse.core.internal.registry.RegistryObjectManager	2	128	>= 7.324.064
java.lang.String[]	35.115	1.293.200	>= 6.976.224
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader	76	5.472	>= 6.811.832
org.eclipse.jdt.internal.ui.text.spelling.SpellCheckEngine	1	24	>= 5.882.240
org.eclipse.jdt.internal.ui.text.spelling.engine.LocaleSensitiveSpe	2	64	>= 5.480.632
org.eclipse.core.internal.registry.ConfigurationElement	16.896	811.008	>= 5.331.656
java.util.ArrayList	18.068	433.632	>= 4.949.920
org.eclipse.jdt.internal.compiler.util.SimpleLookupTable	200	4.800	>= 4.482.184
org.eclipse.core.internal.registry.ExtensionRegistry	2	128	>= 4.409.296
org.eclipse.jdt.internal.core.search.indexing.IndexManager	1	64	>= 3.798.992
org.eclipse.jdt.internal.core.index.Index	150	4.800	>= 3.756.920
org.eclipse.jdt.internal.core.index.DiskIndex	150	12.000	>= 3.684.448
org.eclipse.jdt.internal.compiler.util.HashtableOfObject	102	2.448	>= 3.373.144
char[][]	5.241	445.792	>= 3.183.472
org.eclipse.jdt.internal.core.JavaModelManager	1	160	>= 2.767.744
byte[]	1.158	2.339.672	>= 2.339.672
java.util.Hashtable$Entry[]	765	102.912	>= 2.099.960
java.util.Hashtable	627	25.080	>= 1.993.112
int[]	36.667	1.981.368	>= 1.981.368
org.eclipse.osgi.framework.internal.core.BundleHost	385	21.560	>= 1.806.696
org.eclipse.ui.internal.console.ConsoleDocument	3	312	>= 1.597.720
Total: 29 of 13.193 entries displayed	**1.063.145**	**51.436.256**	

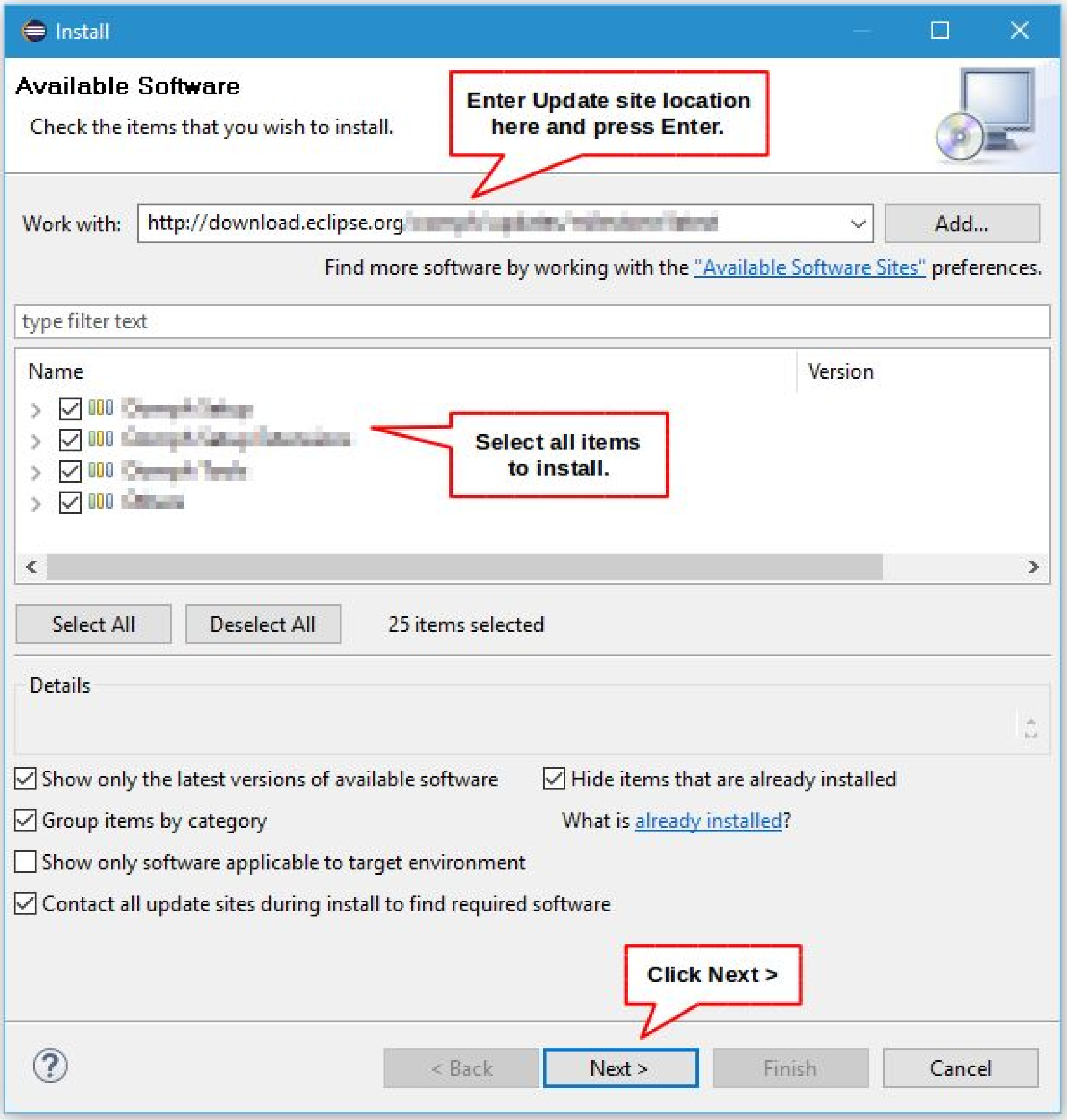
Installation

To install MAT, first ensure that the Eclipse IDE is installed.

Then go to Help > Install New Software…

In “Work with”, insert the update link and download the contents.

E.g.: https://download.eclipse.org/mat/1.4/update-site/

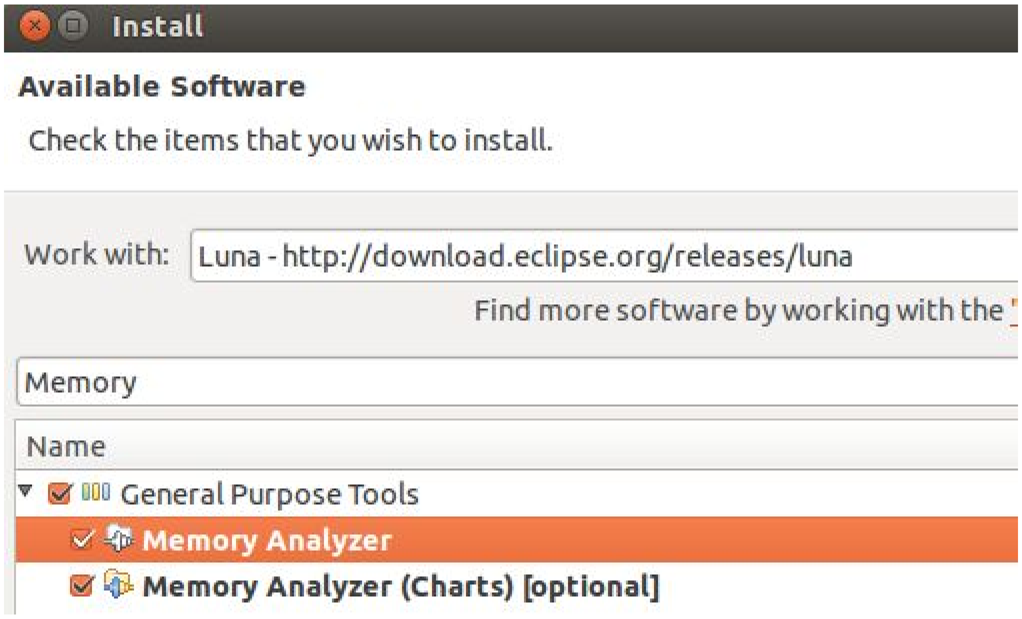

Install
Available Software
Check the items that you wish to install.
Enter Update site location here and press Enter.
Work with:
Add...
Find more software by working with the "Available Software Sites" preferences.
type filter text
Name
Version
Select all items to install.
Select All
Deselect All
25 items selected
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
Click Next >
< Back
Next >
Finish
Cancel

Work with: http://download.eclipse.org

25 items selected

- [✓] Show only the latest versions of available software
- [✓] Hide items that are already installed
- [✓] Group items by category
- [ ] Show only software applicable to target environment
- [✓] Contact all update sites during install to find required software

Select *General Purpose Tools*, *Memory Analyzer* and *Memory Analyzer (Charts)*.


Install

Available Software

Check the items that you wish to install.

Work with: Luna - http://download.eclipse.org/releases/luna

Find more software by working with the

Memory

✓ General Purpose Tools

✓ Memory Analyzer

✓ Memory Analyzer (Charts) [optional]

Utilizing MAT

First, create a heap dump by running the application in Eclipse with the argument:

-XX:+HeapDumpOnOutOfMemoryError

Go to File > New > Other > Heap Dump to set up the heap dump.

Once the heap dump has been created (look for the .hprof ending), open it via double-clicking it.

Open your project and click Leak Suspects Report. Then click Finish.

From there, you can analyze the heap dump and the objects that are used in the program. The dominator tree provides information on the used objects, making it useful to identify memory leaks.

Further Reading

https://www.eclipse.org/mat/

https://www.vogella.com/tutorials/EclipseMemoryAnalyzer/article.html#installation

Memory Analysis Over Time in Ruby

REQUIRED RUBY FILES TO INSTALL:

createdb.rb

db.rb

gemfile

gencsv.rb

genimport.sh

Graph.rb

All of these files can be found at the Github link for this method.

You can use the Ruby command *ObjectSpace* to obtain information regarding objects that may be sticking around. This can be used to dump all objects in memory to a file at spaced time intervals (in the source's example, an interval of 5 minutes is used). The objects that are consistently being dumped to the file are the ones that are causing memory leakage.

To implement the object dump in your program, first you need to include the gem *objspace*. Based upon whether or not you are developing a Rails application, there are two ways to create a dump function. If you are not creating a Rails application, follow Example 1. If you are, follow Example 2.

Example 1 - (Regular Ruby Application File):

NOTE: This method assumes that you have a main event loop that you can call every X minutes you want to dump the data.

Create a function called heap_dump that mimics the following code closely:

```
def heap_dump

    GC.start

    I = Time.now.strftime(‘%s’)

    open(“/tmp/ruby-heap-#{i}.dump, “w”) do |io|

        objectSpace.dump_all(output: io)

    end

end
```

<u>Before you main function code, insert the following:</u>

```
ObjectSpace.trace_object_alloctations_start
```

At the end of your main loop, call the heap_dump function.

For Example:

```
mainloop do

    # put your code here

    heap_dump

end
```

Example 2 - (For Rails Applications):

Add the following code to a controller in your Rails application that you visit every X minutes (the amount of time you want to track between object dumps):

```
class HeapDumpsController < ActionController::Metal

def heap_dump

    if ENV['HEAP_DUMP'] == '1' && params[:token].to_s == ENV['HEAP_DUMP_TOKEN']

        heap_dump

        self.response_body = 'Dumped heap'

    else

        self.status = 401

        self.response_body = 'Invalid token'

    end

end
```

Next, add the following line to your configuration file:

```
get "/heap_dump", to: HeapDumpsController.action(:heap_dump)
```

Finally, add the next code snippet to your heap dump tracing file:

```
if ENV['HEAP_DUMP'] == 1

    require 'objspace'

    ObjectSpace.trace_object_allocations_start

end
```

Now, to analyze the memory leak, take the heap_dump text file you generated and load it into a database. First, initialize the database using the following command in the terminal:

bundle exec ruby createdb.rb

Next, import your data. Importing the data will take some time, so the importing process is split into two steps; conversion to SQL files, and then loading the converted data into the database. Enter the following on the command line to do so:

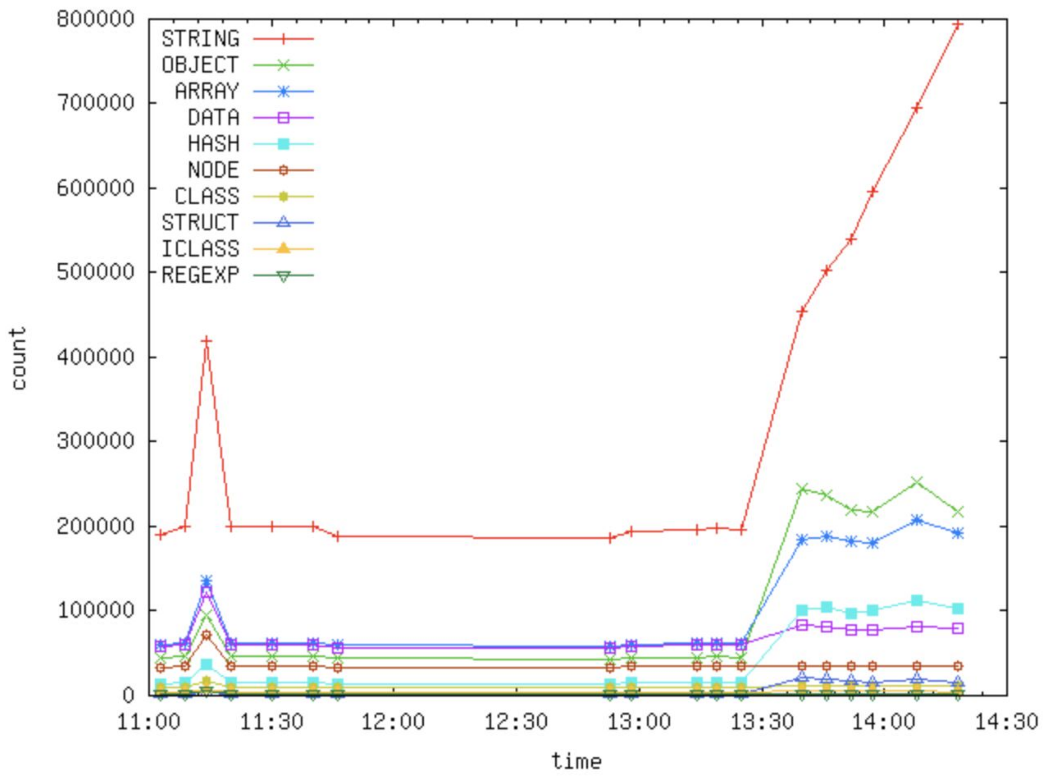
bundle exec ruby gencsv.rb

sh genimport.sh | psql mem_analysis

Once all of the data is imported, then we need to create the graph using the following terminal command:

bundle exec ruby graph.rb type-mem

A graph will be generated with all object types as a line graph each, plotted as Time vs. # of Object Type Count. The types that have a flat slope are not the troublemakers. The object type that has an increasing slope is the one that is causing the memory leak.



If you want to look further into which variable exactly is causing the issue, create another graph of just that object type using:

bundle exec ruby graph.rb [object type]-mem

If you still cannot find the issue, you may need to look into modifying the graphs.rb file to include more cases of how the memory leak could occur.

Further Reading

https://gist.github.com/wvengen/f1097651c238b2f7f11d

Hands-On Exercises

Link to first exercise: https://tcnj.instructure.com/courses/1630707/pages/group-6-activity-1

Link to second exercise: https://tcnj.instructure.com/courses/1630707/pages/group-6-activity-2