

Handling “Out of Memory” Errors

John Tang Boyland

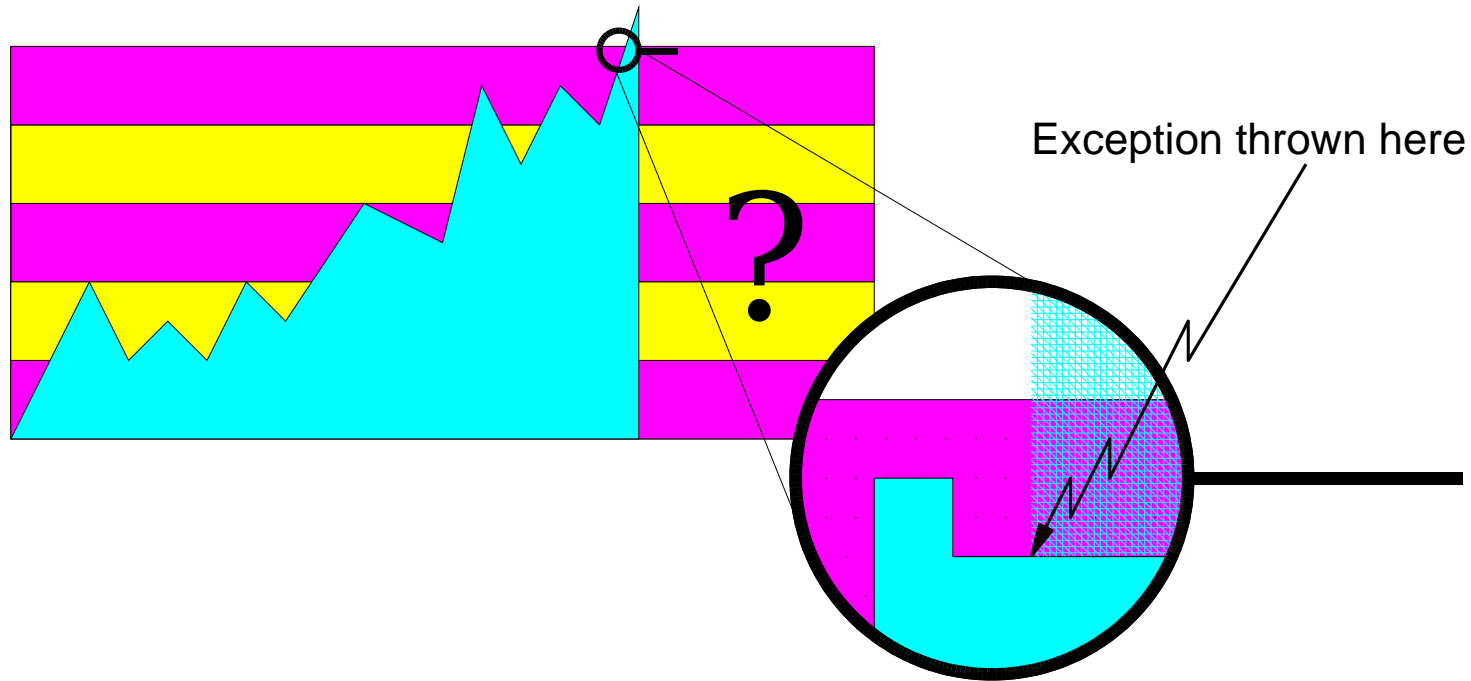
ECOOP EHWS, July 25, 2005



Handling “Out Of Memory” Errors

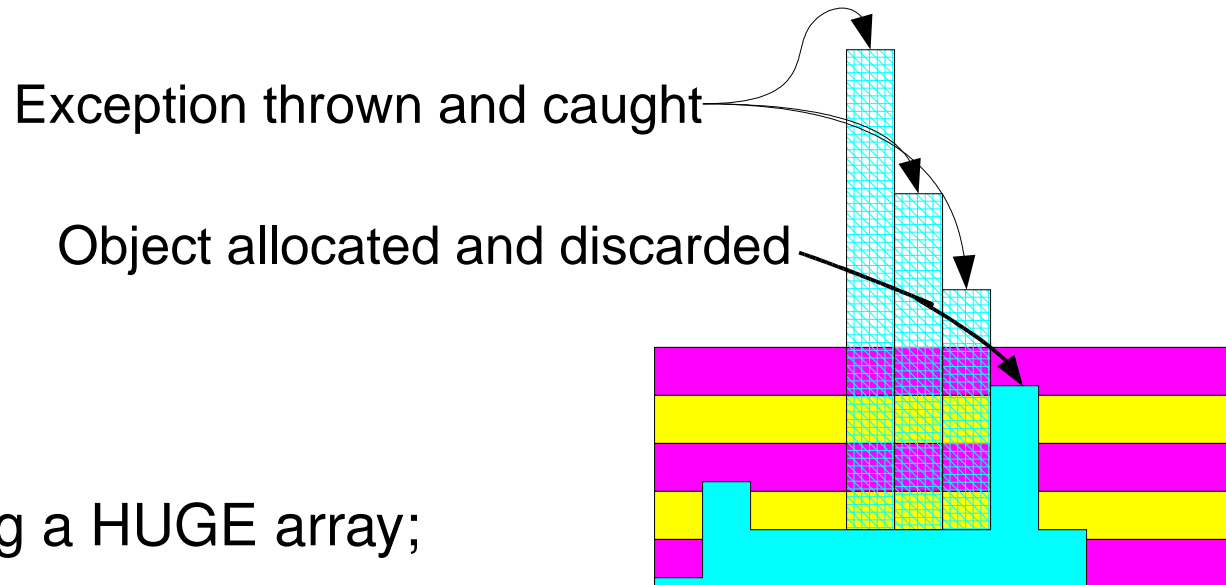
- Current advice for handling `OutOfMemoryError`
 - Use to find memory limits.
 - Don't! There's nothing you can do.
 - ... but Eclipse does!
- The “Hedge” technique.
 - allocate a large hedge;
 - free when recovering from low memory condition.
- Desiderata
 - Language Specification: What is safe if memory is low?
 - Compilers: Don't move allocation later or deallocation earlier.
 - Runtime: per-thread memory restrictions.

An OutOfMemoryError occurs



- Recovery difficult because of “low memory” condition.
- When exception is thrown, last request is *not* fulfilled.

Measuring Available Memory



- A loop:
 - Try allocating a HUGE array;
 - Catch the exception and try again with a smaller amount.
 - Repeat until no exception is thrown.
- A rough underestimate of available memory.
(More accurate than `Runtime.freeMemory()`.)

Reasoning About OutOfMemoryError

- Could occur at any time:
 - even in code “proven” not to raise an exception;
 - `OutOfMemoryError` is a subclass of `Error`,
(in principle) “unpredictable” and “unpreventable” errors.
- ... well, *almost* any time:
 - if memory needed (allocation, boxing, concatenation);
 - if stack needed (call, local var. frame);
 - if exception created (NPE, `ArrayStoreException`, etc).
- Typical advice: don't try to handle it.
- Alternate advice: use soft/weak references.

A “Real” Program Must Handle the Error

- Almost no realistic program can provably avoid running out of memory.
- For example: Eclipse
 - uses more memory if more files are being edited;
 - memory is used by many different parts (GUI, Compiler, assistance, markers etc)
- Crashing on OOME is unacceptable:
 - user’s work is lost, and
 - workbench left (perhaps) in inconsistent state, but
 - logging errors or saving files taken memory;
- The error must be handled.

Handling OutOfMemoryError in Eclipse (1 of 2)

- Eclipse catches OOME and displays warning dialog:
 - but memory is low;
 - dialog appears after emergency exit fails;
 - otherwise only error messages on Unix stdout.
- Eclipse 3.1 uses a larger max heap size than previously
 - Normally degradation (thrashing) long precedes OOME.
 - Artificially lowering the heap size gets previous behavior.

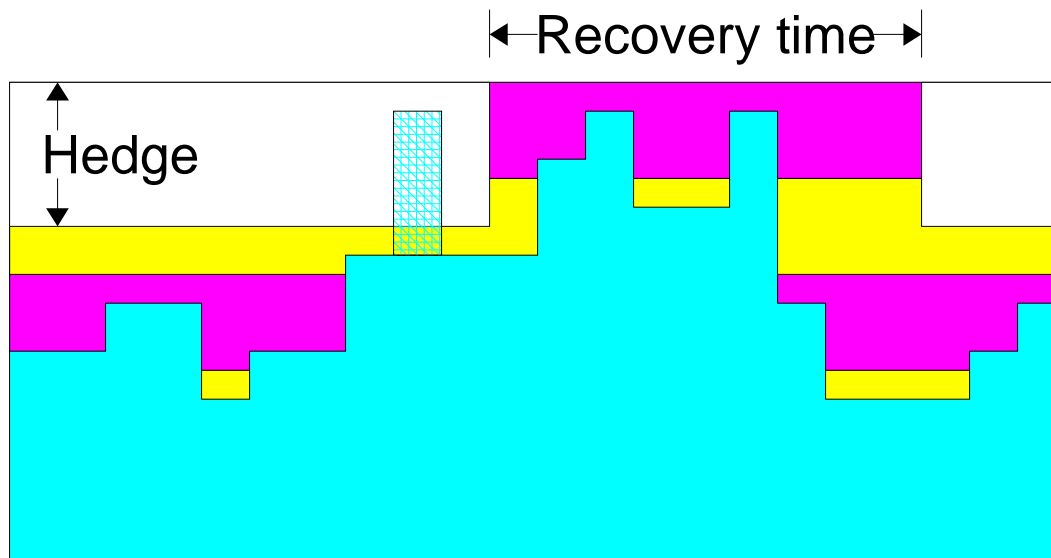
Handling OutOfMemoryError in Eclipse (2 of 2)

```
Exception in thread "...JavaReconciler" java.lang.OutOfMemoryError
Exception in thread "...JavaReconciler" java.lang.OutOfMemoryError
Exception in thread "...JavaReconciler" java.lang.OutOfMemoryError
Error while logging event loop exception:
java.lang.OutOfMemoryError: Java heap space
Logging exception:
java.lang.OutOfMemoryError: Java heap space
Error while informing user about event loop exception:
java.lang.OutOfMemoryError: Java heap space
Dialog open exception:
java.lang.OutOfMemoryError: Java heap space
Fatal error happened during workbench emergency close.
java.lang.OutOfMemoryError: Java heap space
Unhandled event loop exception
Reason: Java heap space
```

○ Then dialog brought up.

The “Hedge” Technique

- Pre-allocate a large area (the “hedge”);
- When `OutOfMemoryError` happens, release it;
- After recovery re-allocate hedge.



Difficulties Using the Hedge Technique

- Need to overestimate memory required for recovery;
- Interrupted computation may leave data inconsistent;
- `finally` clauses before recovery may re-throw OOME;
- Error may be thrown in thread other than the “guilty” one;
- Compiler may move allocation later or deallocation earlier;
- Cannot be made automatic.

(see next slides)

One Problem Leads To Another (1 of 3)

- To avoid corruption, we introduce “finally”:

```
void performAction()  
{  
    start();  
    doIt();  
    cleanup();  
}
```

→

```
void performAction()  
{  
    start();  
    try {  
        doIt();  
    } finally {  
        cleanup();  
    }  
}
```

- But what if `cleanup` needs memory (heap/stack) ?

One Problem Leads To Another (2 of 3)

- So we pre-allocate some memory:

```
void performAction()
{
    start();
    int[] space = new int[1000];
    // Point A
    try {
        doIt();
    } finally {
        // Point B
        space = null;
        cleanup();
    }
}
```

- But what if the compiler ...
 - moves the allocation later (B)?
 - moves the deallocation earlier (A)?

One Problem Leads To Another (3 of 3)

- Fake uses force early allocation.
- Fake tests force late deallocation.

```
void performAction()
{
    start();
    int[] space = new int[1000];
    space[45] = 1+space[fact(6)];
    try {
        doIt();
    } finally {
        if (space[45] > space[44]) {
            space = null;
            cleanup();
        }
    }
}
```

- We have obfuscated our program.

Placing Hedge Recovery

- At outer level
 - + few code changes;
 - + lock state clear;
 - work undone;
- Close to allocation
 - + recovery fast;
 - state unclear;
- If automatic, then how is recovery invoked?
 - at error point, then re-entrancy problems;
 - elsewhere, then `finally` is still an issue.

Experiences With Hedge Recovery

- Importing Java Into Internal Representation:
 - Must persist in “eras”;
 - As few eras as possible;
 - No easy way to use weak/soft references;
- Converted JDK 1.4.2 provided source
 - 4500 source files;
 - 12 hours;
 - 11 `OutOfMemoryErrors` generated;
 - (300 Mb max heap on Solaris x86).
- Avoided threading issues (single-threaded code).

Conclusions

- Hedge recovery can work.
Perhaps Eclipse could use it.
- Hedge recovery would be safer if:
 - Language specified what operations need memory;
 - Compilers don't move allocation/deallocation past `try-finally` boundaries;
 - Threads had own memory restrictions.
- Thrashing is a good alternative for interactive programs.