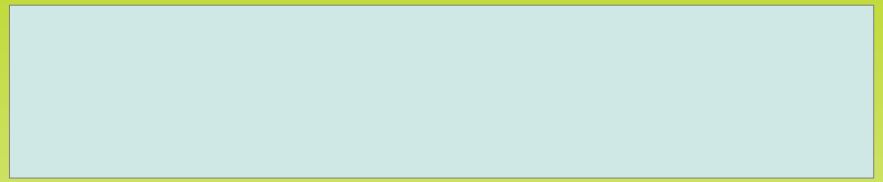
Samsung Developers Conference 2014

Top Ten Memory Management and Tuning Tips

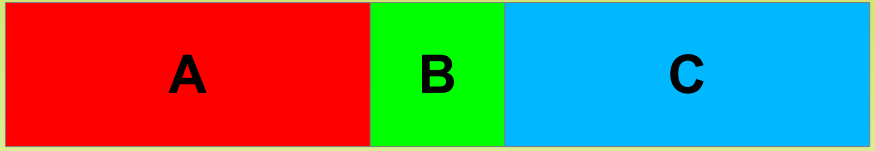
#1: Understand the Problem

- Symptom: Crashing on a Large Allocation
  - Example: loading a bitmap
- Cause: **Heap Fragmentation**
  - Dalvik VM does not coalesce free space, unlike Java VM
  - Eventually, you no longer have access to a large enough free block for your allocation

A “Moving” Garbage Collector



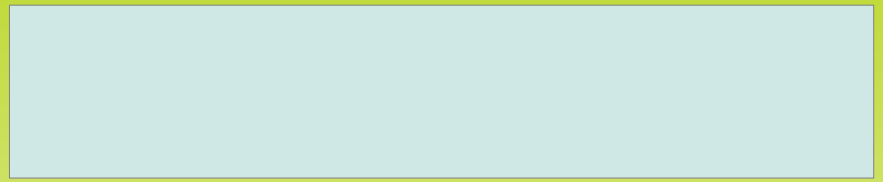
In the beginning, there was heap, and it was good...



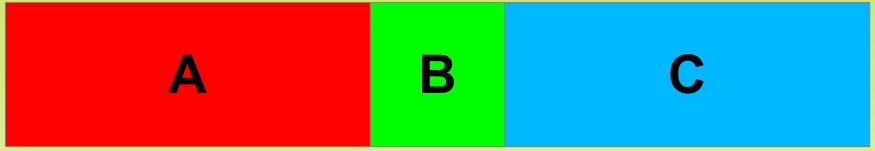
The story of “The Three Little Blocks and the Big Bad OutOfMemoryError”

Trash compactor

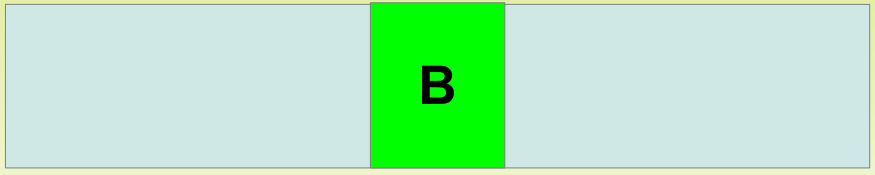
Dalvik: Not Moving Quite So Much



In the beginning, there was heap, and it was good...



The story of “The Three Little Blocks and the Big Bad OutOfMemoryError”



Where, o where has my heap space gone? O where, o where can it be?

#1: Understand the Problem

- Symptom: Crashing on a Small Allocation
  - Example: creating a fairly ordinary object
- Cause: **Heap Exhaustion**
  - Most objects are fairly small
  - If you cannot allocate one of those, you truly are out of memory

"We have to make *this* fit into the hole made for *this*, using nothing but *that*."

(from Apollo 13, 1995)

#2: Design for Low RAM

- Focus on the Art of the Possible
  - ...and get to the impossible sometime after lunch
- Examples
  - Massive scrolling lists
  - Pre-fetching content to caches
  - Panning through huge images

#3: Design For Even Lower RAM

- Not All Devices Are Created Equal
  - Example: Android One
- Graceful Degradation/Progressive Enhancement
  - Work for the low end
  - Work well for the high end
- Example: Maps

#4: Grok Your Heap

- Key Questions
  - How much heap are you using?
  - How many instances of your classes are floating around?
    - ...and why are they still there? Shouldn't they have gone home by now?
  - What are your objects holding onto?

Head to the MAT

- Memory Analysis Tool (MAT)
  - Used to examine heap dumps
    - From DDMS
    - From Debug#dumpHprofData()
  - Available as standalone GUI or as Eclipse plugin
  - Uses
    - Understanding what's using up your heap
    - Finding memory leaks

Head to the MAT

- Know Your Heap
  - What are you allocating and holding onto long-term?
  - IOW, what are your key GC roots?
    - Static data members
    - Threads
    - Long-running services
  - What can you do to minimize what they reference?

Head to the MAT

- Look for Leak Candidates
  - Instances of your classes that should not be around
  - Bitmaps and byte arrays
- Trace Your GC Roots
  - “GC Roots” = “what the @#$%&! is holding onto this &%&$ thing?!?”

#5: Allocate Only What You Need

- Understand the Use of the Data
  - “Data” = bitmaps, database contents, etc.
- Find Ways to Obtain the Required Subset of the Data
  - “Required” = what you need now, more than what you might need sometime later

Ponder Bitmap Sizes

- Load the Size You Need
- Example: ListView rows
  - Load thumbnails, using `inSampleSize` on `BitmapFactory.Options`
  - Only load full-size image for the ones the user clicks upon and brings up some sort of details fragment/activity/whatever

Ponder Bitmap Pixels

- Default: ARGB_8888
  - 4 bytes per pixel
- Alternative: RBG_565
  - 2 bytes per pixel = half the memory for same resolution
  - No transparency
- Example: ListView rows
  - Thumbnails perhaps can get away with less color depth

#6: Avoid Leaks

- “Leak” = Unintended Retained Data
  - “Retained” = reachable from a root
    - Static data member
    - Thread
    - Application singleton, ContentProvider, etc.
  - “Unintended” = “gosh, that's a lot of stuff”

Libraries Over DIY

- Common Subsystems That Might Leak
  - Image Loaders
  - ORM
  - Event Buses
- Use Something Tested Over Something New

Choose the Right Context

- Beware of a Custom Application Class
  - Singleton, so anything it holds onto cannot be GC'd
- Use Application in the Right Places
  - If you are holding onto things in static data members that might be associated with a Context, use Application, as it is “pre-leaked”
  - Failure to do so: leak activities, etc.

Watch Your Threads

- Threads = GC Roots
  - Anything reachable by a thread cannot be GC'd
- Tips
  - Leaking threads = leaking heap
  - Ensure threads in pools `null` out data members
  - Beware the everlasting service!
    - Its threads, and anything else, cannot be GC'd
    - Also screws up multitasking, etc.

#7: Scale All Your Caches

- Use `ActivityManager` and `getMemoryClass()`
  - Plus algorithm for capping your caches to certain portion of possible heap
  - On top of using weak or soft references
- Beware Multiple Caches
  - Library A has a cache, Library B has a cache, etc.
  - Sum of all caches must be reasonable

Clean Up with onTrimMemory()

- Called On Your Components
  - Activity, Service, ContentProvider, Application, etc.
- Objective: Release Some Heap Space

Clean Up with onTrimMemory()

- You Are Safe (But Please Be Kind to Others)
  - TRIM_MEMORY_RUNNING_MODERATE
  - TRIM_MEMORY_RUNNING_LOW
  - TRIM_MEMORY_RUNNING_CRITICAL
- You Are Invisible
  - TRIM_MEMORY_UI_HIDDEN

Clean Up with onTrimMemory()

- Your Time is Running Out
  - TRIM_MEMORY_BACKGROUND
  - TRIM_MEMORY_MODERATE
  - TRIM_MEMORY_COMPLETE
    - Also available as onLowMemory() for sub-API Level 14 devices

#8: Recycling is Good For Your Environment

- Object Pools
  - Collection of some common resource (objects, threads, etc.)
  - Access patterns to acquire and release
  - Pre-allocated minimum pool contents
  - Cap on maximum pool size
    - Grow as need to limit, release resources to shrink
    - Acquire blocks if needed for other thread to release

(Not) Everybody Into the Pool

- Good News!
  - Avoid heap fragmentation!
  - Slight heap compacting effect via pre-allocation
  - Reduce GC processing time
  - Minimize constructor CPU time
- Bad News!
  - Shades of `malloc()` and `free()`

Use inBitmap

- `BitmapFactory.Options` field
- Specifies `Bitmap` to reuse
  - API Level 19+: must be same size or larger than bitmap to be loaded
  - API Level 18 and lower: must match exactly
- Great for thumbnails, other scenarios with constant bitmap sizes

#9: Let Your Process Go

- Nuke Your Entire Heap From Orbit
  - It's the only way to be sure that you're going to get a nice clean heap again
- Avoid the “Must Keep Process Running” Syndrome
  - Or use a second process for focused heap for long-running stuff (but that's cheatin')

#10: Cheat with Integrity

- “Cheat” = Cause Problems Elsewhere to Solve Heap Pressures
- “with Integrity” = Use as Stop-Gap
  - Have a plan for reverting the cheating

Cheat: Request Large Heap

- `android:largeHeap` in `<application>`
- **Probably** gives you a larger heap on API Level 11+
  - Depends a bit on device capabilities
- Use `getLargeMemoryClass()` to determine how large your heap is

Cheat: Multiple Processes

- Reason #1: More Heap
  - Use second process for specific memory-intensive operations
  - Workaround for pre-Honeycomb devices
- Reason #2: Focused Heap
  - Use second process for long-running background services
  - May be able to accomplish same basic end with `onTrimMemory()`

Cheat: NDK

- Native allocations do not count against Dalvik heap
- Use native code for memory-intensive operations
  - Particularly where you could gain some performance from native code
  - Example: image processing

The Costs of Cheating

- Larger system RAM consumption
  - More likely to get blamed by OS, users
  - More likely to have background process terminated
- Multiple processes = IPC
  - CPU and battery consumption
  - Keep your protocol relatively coarse-grained

The ART of Memory Management

- ART = Android Runtime for 5.0+
- Moving Garbage Collector
  - When we're in the background, to coalesce free space
- Large Object Space
  - Faster releasing of unused bitmaps
- Net = Fewer OutOfMemoryError messages

Summary

- Understand the Problem
- Design for Low RAM
- Design for Lower RAM
- Grok Your Heap
- Allocate Only What You Need
- Avoid Leaks
- Scale All Your Caches
- Let Your Process Go
- Recycling is Good for Your Environment
- Cheat with Integrity