Java Memory Management

This article provides an overview on Java memomy management.

Memory management can be one of the great joys of developing with a garbage-collected language like Java. Being able to create objects temporarily, without having to worry about making sure they are destroyed later, makes object orientation much more usable. Using libraries without having to worry about who is responsible for allocating and deallocating memory means less time rummaging through documentation, and less time tracking down memory leaks.

In the world of Java ME, however, we're often working in very tight spaces, sometimes as low as 200k, so memory management is still something we have to think about. And it can still be something we have to *do*.

Which means, you'll need to know how memory is being managed...

The Heap(s)

The heap is the space used for all runtime data. This can include classes and objects, as well as *stack frames*. Stack frames are used by methods to store parameters and local variables, as well as temporary values during expression evaluation. They are created when a method is entered, and destroyed when it exits. I say "destroyed", what I really mean is "becomes garbage".

Heaps come in all shapes and sizes, which can make life confusing for those new to mobile Java.

Every implementation has at least one heap. This is the main place where classes are loaded and objects are allocated. It might be a fixed size (like Series 40), or it might grow as required (S60). The Runtime class has methods for finding the size of this heap and how much is free (unallocated).

Some implementations will use additional heaps. These will usually be specialised in some way. Examples include:

- Native object heap: sometimes, Java objects are used to access "native objects" - objects that are created and managed by the operating system or user interface. In some cases, these objects will exist outside of the Java heap, with a small object inside the Java heap that interfaces between the two environments. Images and 3D objects can fall into this category.
- Image heap: images heaps are used only for images. They are sometimes more closely coupled to the video hardware to improve graphical performance.
- Large object heap: this kind of heap is used for "large objects" - which could include images or large arrays.

Heaps like these may be shared between applications. There may be no way to find out how large these heaps are, or how much is available. Heaps shared between applications can be a common cause of OutOfMemoryErrors that occur only the first time the application is executed after download, usually because the browser is still consuming memory.

Since some kinds of object may have to be stored in a specific heap, running out of space in any one heap can result in an OutOfMemoryError.

Garbage Collection

The garbage collector (GC) gets rid of objects (from all heaps) that you don't need anymore. Eventually. It doesn't get rid of them immediately that they become unreachable.

What you can guarantee, is that the GC will have done it's best to find memory *before* an OutOfMemoryError is thrown. Usually, the GC runs when there is not enough memory to create a new object. If you watch the memory monitor of an emulator, you'll usually see the free memory drop steadily towards zero, then bounce back up as the GC does its work.

It will never release an object that you can possibly use. If any variable anywhere in the system refers to an object, then the object isn't garbage.

There doesn't always need to be a variable to keep an object. An object isn't garbage if it's referenced from a stack frame. For example:

```
String s2 = s1.trim().toLowerCase();
```

The call to trim() creates a new String object. A reference to this String is placed on the stack so that the toLowerCase() method can be invoked on this new object. toLowerCase() creates another object, a reference to which is saved in the variable "s2". Once toLowerCase() returns, the object created by trim() is no longer needed. Its reference is lost, and the object becomes garbage.

Some objects are referenced by system objects.

- Threads are referenced by the Java Virtual Machine, as long as they're running. So, threads can't become garbage until they die.
- Each MIDlet is associated with an instance of Display. That object never becomes garbage (until the application terminates). Any objects reachable through the Display won't become garbage either. The Display holds a reference to a Displayable object - whatever you passed to Display.setCurrent(). That Displayable might also reference object, such as Images used on a Form or List.

You can, you're probably aware, invoke the garbage collector manually, by calling System.gc(). In general, this is a bad plan. When the GC runs automatically, it might (on some implementations) run a quick cycle, freeing up enough memory for the program to continue. But the rules say that when you run it manually, it must make every attempt to collect *all* the garbage. This can be much slower on devices with large heaps. S60 devices are particularly prone to this.

On most devices, the garbage collector will defragment the available space in the heap. This ensures that you can create objects as big as possible. However, some older devices don't. This can result in the free space being split into many small pieces. The largest object you can create is restricted to the largest single piece, so you can run out of memory even if you have enough memory *in total*. On devices like this, running the GC manually after creating a lot of garbage can reduce fragmentation. It's also useful (on the same devices) in-between loading images.

Manual GC's are best performed when the user will expect some kind of delay - like, when you have a "loading" or "please wait" screen displayed.

What The Garbage Collector Doesn't Do

It doesn't kill threads. Threads die when their run() method terminates, either by hitting the closing brace, reaching a return statement, or throwing an exception.

It doesn't close files, network connections, record stores, media players, or anything else. These objects **must** be closed by calling the appropriate methods, or scarce system resources might not get released.

"But, I Don't Create Any Garbage"

It's pretty much impossible to execute Java code and not create any garbage. Even if you don't create objects, library code will do. Graphics objects will be created every time the screen is re-painted, for example.

Since trying to avoid garbage is futile, you are generally better to create new objects when you need them, and not to try to re-use existing objects. Objects that have already been used might retain unwanted data, or be left in a different state than a fresh object, resulting in peculiar bugs.

Classes

The code you run gets loaded into the heap. Classes are loaded as needed, not necessarily all at the start. Classes are never unloaded (until the VM terminates), because they never become garbage. (In Java ME CLDC, that is. Classes *can* become garbage in versions of Java that support multiple class-loaders, like J2SE or ME CDC.)

In most cases, this doesn't include system classes. System classes are usually pre-loaded in the phone's firmware.

Objects

Arrays are objects too, so all the same rules apply to them.

Since the Java Virtual Machine specification doesn't describe how objects should be represented in memory, it's impossible to say how much memory things *actually* take. But, based on experience, here's some pretty close figures.

On all the VM's I've tried, all objects take 16 bytes, no matter what. This is the overhead that the VM needs to manage the object. Add to that whatever is required for their fields, and round that up to a multiple of four (memory is almost always allocated in four-byte chunks).

Object references, floats and ints need four bytes each. Bytes need (you guessed) one byte each. So do booleans. Shorts and chars need two. Longs and doubles need eight.

A couple of examples will help to put this in some perspective, and maybe explain why you can seem to run out of memory a long time before you expect to.

Consider this.

```
byte[][] array = new byte[3][3];
```

An array big enough to hold nine bytes. How much heap will that need? Well, a two-dimensional array is actually an array of arrays (an array of objects, since arrays are objects). So, the following code is equivalent.

```
byte[][] array = new byte[3][];
array[0] = new byte[3];
array[1] = new byte[3];
array[2] = new byte[3];
```

The first array we create is an object, so needs 16 bytes of memory. On top of that, it contains three object references of four bytes each. I make that 28 bytes.

Then we create another three objects. These also need 16 bytes, plus enough space for the three bytes each one contains. That's going to mean *four* bytes, since it has to be a multiple of four. Three objects of 20 bytes each, so 60 bytes.

That's 88 bytes in total. Just to store 9. Small objects can be very, very expensive.

Strings are also deceptively expensive.

Each String is an object (16 bytes of overhead), containing an int (four bytes, to store the length of the String) and a reference to a char[] (four bytes). The char[] is also an object (16 bytes), needing two bytes for each character, rounded up to a multiple of four. That's 40 bytes, plus two per character, plus zero or two bytes of padding. The String "Hi!", for example, takes 48 bytes.

Another String example: let's say you have a dictionary of 15,000 words, averaging seven characters per word. That's 105,000 characters. The easy trap to fall into is to think that this will need 105,000 bytes.

A seven-character String takes 56 bytes. 15,000 * 56 comes to 840,000 bytes.

You'll also need somewhere to store the references to these objects. A String[] of 15,000 elements will need a little over 60,000 bytes, taking the total to over 900,000 bytes. If you want to reference your Strings through something more exotic, like a Hashtable, the total is going to be over 1Mb.

Further Reading

- Memory Usage Images: describes how memory is used for storing images.
- How to use an optional API in Java ME: describes the class loading process, and how to control which classes are loaded and when.