CINCOM Technical Note

VisualWorks Memory Management

Software Release 7.2
Lat updated: October 21, 2003

This document describes the memory management strategies used by the VisualWorks virtual machine (object engine).

This information is helpful when performance tuning certain memory-intensive applications. It is especially relevant for deploying applications with large working sets (e.g. one gigabyte or larger).

The facilities and policies described here are subject to change from release to release, so use this information with caution.

This technical note explores the following topics:

- Memory Layout
- Facilities for Reclaiming Space
- Managing the Object Memory
- Preparing for Deployment
- Implementation Limits

Memory Layout

At start-up, VisualWorks asks the operating system to allocate a portion of the available address space to house objects, native code and other resources, and then begins executing. Subsequently, VisualWorks may grow or shrink its memory usage dynamically.

VisualWorks runs as an operating system process with access to the full 32-bit address space made available to it by the operating system. As VisualWorks uses 32-bit pointers for Smalltalk objects, it can use as much of the 32-bit address space for objects as the host operating sytsem will allow, but on 64-bit operating systems only 32 bits of the address space can be allocated for objects (this limitation may be removed in a future release).

VisualWorks makes a number of demands upon the address space. For example, each of the following can consume a fair amount of memory in the address space:

- Code and static data that belong to the object engine
- Dynamic allocations made by the “C” run-time libraries (such as **`stdio`** buffers)
- Dynamic allocations made by the window-system libraries
- Static and dynamic allocations made by the object engine

This section discusses only the algorithms associated with the last item.

The object engine manages two types of memory space: (1) a set of fixed-size spaces associated with the object engine and (2) a set of variable-sized spaces that comprise the Smalltalk object memory.

Fixed-Sized Spaces

The object engine allocates the following fixed-size memory spaces at system start-up time:

- Compiled Code Cache
- Stack Space
- New Space
- Large Space
- Perm Space

Variable-Sized Spaces

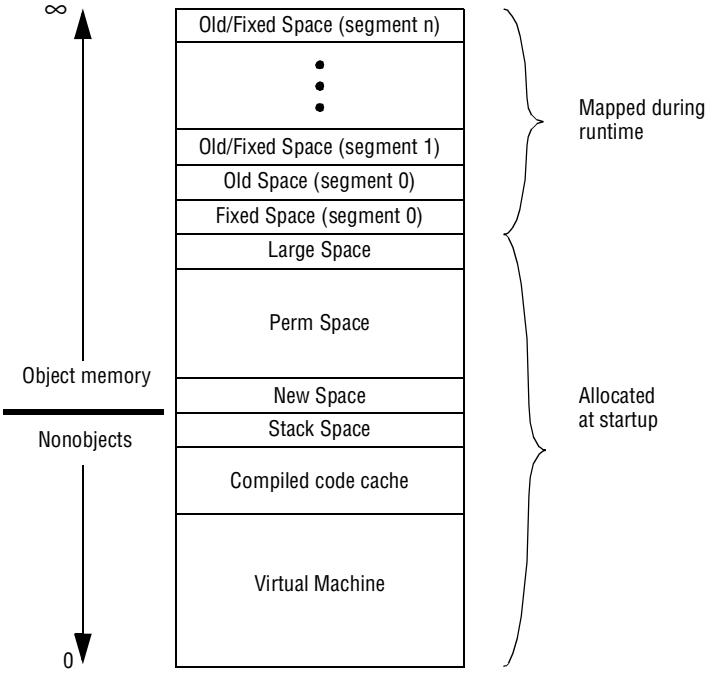
The object engine allocates an initial size for the following variable-sized spaces at start-up, sufficient to hold the existing old and fixed space objects, plus a free-space overhead in each.

- Old Space
- Fixed Space

Memory Organization

Each of these fixed- and variable-sized spaces is used by the object engine to house program elements of a particular type.

The object engine organizes these spaces as shown below:



The diagram shows organization of the address space that belongs to a VisualWorks process, with code, non-objects, and the fixed portions of the object memory in the lower portion of the space, and the dynamic spaces and segments of the object memory above.

Compiled Code Cache

To avoid the overhead of interpreting bytecodes, the object engine compiles each Smalltalk method into the platform's machine code before executing it. The compilation is automatic and transparent to the user.

When a Smalltalk method is invoked for the first time, the object engine compiles it and stores the resulting machine-code in the Compiled Code Cache, so that it can be executed. Once executed, this method's machine-code is left in the Compiled Code Cache for subsequent execution.

As its name suggests, this space is only used as a cache. If the cache begins to overflow, those methods that have not been executed recently are flushed from the cache. This approach gives Smalltalk much of the speed that comes with executing compiled code, most of the space savings, and all of the portability that come with interpretation.

The size of this cache varies, depending on the density of the platform's instruction set. Default sizes are 640 KB for platforms with CISC-based processors and 1152 KB for RISC platforms. These sizes are large enough to contain the machine-code working sets of most applications. The size can be changed at at image startup, as described under "Setting Object Engine Space Sizes" (below), to a maximum of 16 MB.

The compiled code cache is also used by the garbage collector to store the *mark stack* (for details, see "Global Garbage Collector" below), and to decompress compressed virtual image files on start-up.

Stack Space

Each process that is active in the virtual image (VI) is associated with a chain of *contexts*. Contexts are stored in two forms: the standard object format and the frame format.

When a Smalltalk program tries to send a message to or access an instance variable of a given context, that context must be in standard object form and housed in object memory. If it is not already in standard object form, then it is converted. The conversion to and from standard object format is transparent to the user.

On the other hand, when the method associated with a given context is actually being executed, that context must be in frame format and housed in the Stack Space. Once again, the conversion to and from this form is automatic and transparent to the user. The frame format of the contexts has been designed to mate well with the typical machine's subroutine-call instructions.

The Stack Space is used as a cache. If there isn't enough room in the Stack Space to store all of the contexts of all of the active processes, then the object engine converts some of these contexts to standard object form and places them in object memory to clear some Stack Space. When the system needs to execute the methods associated with these contexts, it converts the contexts back to frame format and places them back in the Stack Space.

The default size of this space varies from 20 KB to 40 KB, depending upon whether a given platform handles interrupts in another region of memory, or whether it needs to handle these interrupts in Stack Space. The size can be changed at image startup, as described under "Setting Object Engine Space Sizes" below. You can reduce the size of the Stack Space, at the cost of forcing the object engine to convert contexts more frequently from frame format to standard object format and back again. Or you can increase its size, at the cost of the additional memory.

New Space

New Space is used to house newly-created objects. It is composed of three partitions: an Object Creation Space, which we call Eden, and two Survivor Spaces.

When an object is first created, it is placed in Eden. When Eden starts to fill up (i.e., when the number of used bytes in Eden exceeds a threshold known as the *scavenge threshold*), the system's scavenging mechanism is invoked. Objects that are still reachable from the system roots are placed in whichever Survivor Space happens to be unoccupied at the time (one is always guaranteed to be unoccupied). Thereafter, objects that survive each scavenge are shuffled from the occupied Survivor Space to the unoccupied one. When the occupied Survivor Space begins to fill up (i.e., when the number of used bytes in the occupied Survivor Space exceeds a threshold known as the *tenure threshold*), the oldest objects in Survivor Space are moved to a special part of object memory called Old Space. When an object is moved from New Space to Old Space, it is said to be tenured. Both the scavenge threshold and the tenure threshold can be set dynamically (see the class ObjectMemory for details).

The default size of Eden is 300 KB, and each Survivor Space (they are always identical in size) is 60 KB. These sizes can be changed at at image startup, as described under "Setting Object Engine Space Sizes" (below).

Large Space

Large Space is used to house the data of large byte objects (bitmaps, strings, byte arrays, uninterpreted bytes, etc.). By “large,” we mean larger than 1 KB.

When a large byte object is created, its header is placed in Eden and its data in Large Space. This arrangement permits the scavenger to move the object’s header from Eden to a Survivor Space without having to move the object’s data. In fact, the data that is housed in Large Space is only moved when Large Space is compacted, as part of a compacting garbage collection or to make room for another large byte object or in preparation for a snapshot.

Of course, the data of any object can be housed in Large Space, but small objects and large pointer objects are only placed in Large Space if there is no other place to house them. The data of large pointer objects is not housed in Large Space because it would take up valuable space without saving the scavenger much work. (Since such data is composed of object pointers, the scavenger has to scan it anyway, and it’s not expensive to move the data while scanning it.)

If there are too many large objects to fit in Large Space, older ones are moved to object memory proper.

When the amount of data housed in the Large Space exceeds a threshold known as the LargeSpaceTenureThreshold, the scavenger is informed that it should start to tenure the headers of large objects. During the next scavenge, the headers of the oldest large objects are tenured to Old Space. However, the data of these large objects will not be moved from Large Space until the allocator actually runs out of space in Large Space. Only at that time will the data of these older large objects be moved to Old Space. The LargeSpaceTenureThreshold can be set dynamically.

The default size of Large Space is 200 KB. Again, the size can be changed at image startup, as described under “Setting Object Engine Space Sizes” (below).

Perm Space

Perm Space is used to hold all semi-permanent objects. Because they are rarely ready to die, the objects housed in Perm Space are exempt from being collected by any of the reclamation facilities other than the global garbage collector. By removing such objects from Old Space, the time required to reclaim the garbage that may be present in Old Space is reduced many times.

In the delivered virtual image, most of the objects in the system are housed in Perm Space. Newly created objects that are placed in Old Space by the scavenger are not automatically promoted to Perm Space.

Developers can move Old Space objects into Perm Space (and thus improving the efficiency of garbage reclamation) by creating an image by choosing **File → Perm Save As…** in the VisualWorks Launcher window.

For details, see "Promoting Objects to Perm Space" (below).

Smalltalk Object Memory

In addition to the above fixed-size memory spaces, the object engine also manages two variable-size spaces known as Old Space and Fixed Space. These spaces are warehouses for all objects that are not housed in one of the fixed-size spaces described above.

Old Space

Unlike the above spaces, however, the size of Old Space is not frozen at startup time. Instead, it is configured at startup time with a default of 1 MB of free space. When Old Space begins to run short of free space, the system has the option of increasing its size. This growth is accomplished by means of a primitive that attempts to acquire additional address space from the operating system.The decisions regarding when to grow Old Space and by how much are controlled by an instance of MemoryPolicy. See that class for the default policy.

Although Old Space may be thought of as a single contiguous chunk of memory, it is implemented as a linked list of segments occupying the upper portion of the system's heap. Old Space's growth capability dictates this approach because, for example, I/O routines frequently allocate portions of the heap for their own use, creating intervening zones that divide Old Space into separate segments. In a growing system, then, Old Space may be composed of multiple segments. When these multiple segments are written out at snapshot time, they are stripped of their free space to conserve disk space. In addition, to avoid fragmentation, the segments are coalesced into one large segment when the snapshotted image is loaded back into memory at startup time.

Each Old Space segment is composed of two parts: an *object table* (OT) that is used to house the old objects' headers, and a *data heap* that is used to house the objects' data. The data heap is housed at the bottom of the segment and grows upward; the object table is housed at the top of the segment and grows downward. Both the object table and the data heap are compacted by the compacting garbage collector.

Since the object table and the data heap grow toward each other (thereby consuming the same block of contiguous free space from different directions), the system should never run out of space for new object headers while still having plenty of space for object data, and vice versa. Nor is there any arbitrary limit on the total size of Old Space, the total size of a given Old Space segment, or the number of Old Space segments that can be acquired. The only memory-related resource that the system can run out of is address space. On real-memory machines, this translates to available real memory. On virtual-memory machines, it corresponds to available swap space.

In addition, the system maintains a threaded list of free object table entries and a threaded free list of free data chunks. The incremental garbage collector recycles dead objects by placing their headers and bodies on these lists, and the Old Space allocator tries to allocate objects by utilizing the space on these lists before dipping into the free contiguous space between the object table and the data heap of each segment. Finally, a certain portion of the free contiguous data is reserved for use by the object engine to ensure that it can perform at least one scavenge in extreme low-space conditions, thereby providing the system with one final opportunity to take the appropriate action.

The default size of Old Space is set to the amount needed to house the old space objects in the image plus 1 MB of free space "headroom". The headroom size can be changed at image startup, as described under “Setting Object Engine Space Sizes” (below).

Fixed Space

Fixed Space is used to hold data (the body) of byte objects whose data must not move. This is a requirement for data passed through the threaded API (THAPI), since threaded calls may be in process concurrently with a garbage collection. The data does not move during the object’s life, but the space is reclaimed when the object is garbage collected.

Since the contents of Fixed Space can’t move, it cannot be compacted, and so quickly becomes fragmented. Fixed Space is coallesced at image start-up, so can be compacted by saving the image, quitting, and then restarting.

New Fixed Space segments are added as needed, like Old Space. When the image is saved, these multiple segments are stripped of their free space to save file space. They are then coalesced into one large segment when the image is loaded back into memory at startup.

Object data ends up in fixed space if it is either:

- allocated explicitly, or
- passed as an argument of a threaded call.

The default size of fixed space is 200 K. Again, the size can be changed at image startup, as described under “Setting Object Engine Space Sizes” (below).

Remembered Tables

Remembered Tables are structures used by the garbage collector to track references between various spaces. They are housed in Old Space.

The *remembered table* (RT) is a special table that contains one entry for each object in Old Space or Perm Space that is thought to contain a reference to an object housed in New Space.

The objects in the remembered table are used as roots by the scavenger — if an object is not transitively reachable from either the remembered table or the Stack Space, it will not survive a scavenge. The remembered table is expanded and shrunk as needed by the object engine. It is expanded if the object engine tries to store more entries than the RT can currently house, and it is shrunk during garbage collections when it has become both large and sparse, which can occur if a large number of entries were added and subsequently removed.

The *old remembered table* (OldRT) is a special table that contains one entry for each object in Perm Space that is thought to contain a reference to an object housed in Old Space or Large Space.

The objects in the OldRT are used as roots by the incremental garbage collector and the compacting garbage collector — if an object is not transitively reachable from the OldRT, it will not survive a garbage collection. The OldRT is expanded and shrunk as needed by the OE. It is expanded if the OE tries to store more entries than the OldRT can currently house, and it is shrunk during garbage collections when it has become both large and sparse, which can occur if a large number of entries were added and subsequently removed.

Facilities for Reclaiming Space

The object engine has several facilities for reclaiming the space occupied by objects that are no longer accessible from the system roots:

- Generation scavenger
- Incremental garbage collector
- Compacting garbage collector
- Global garbage collector
- Data compactor

Except for the scavenger, the object engine does not invoke these facilities directly. Policy decisions such as these are controlled at the Smalltalk level — see the ObjectMemory and MemoryPolicy classes for the default policies.

Generation Scavenger

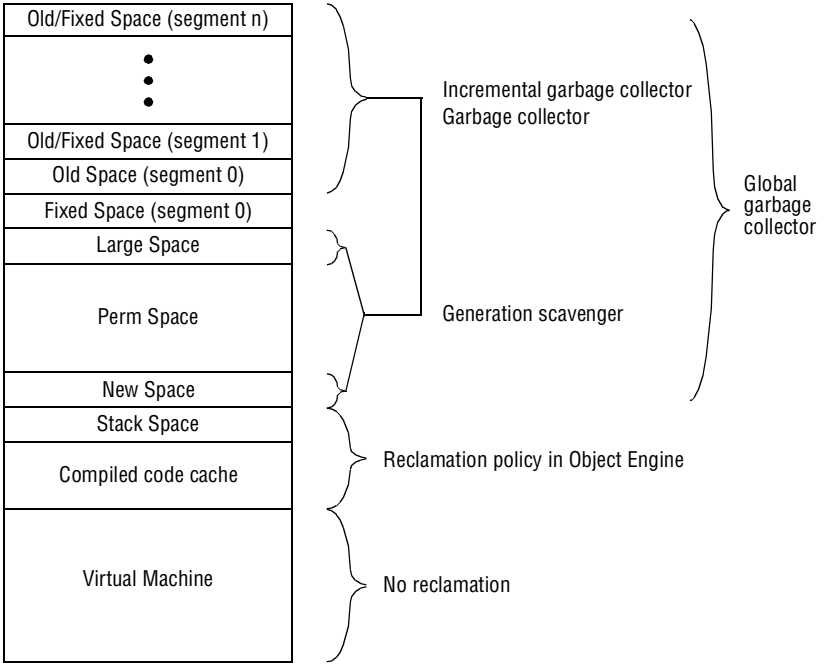
The primary reclamation system is a *generation scavenger*. The scavenger flushes objects that expire while residing in New Space (which typically applies to more than 95 percent of objects).

Briefly, the scavenger works as follows. Whenever Eden is about to fill up, the scavenger is invoked. It locates all of the objects in Eden and the occupied Survivor Space that are reachable from the system roots. It then copies these objects to the unoccupied Survivor Space. Once this copying is done, Eden and the formerly occupied Survivor Space contain only corpses — they are effectively empty and can be reused. The scavenger uses the objects in the remembered table and the objects referenced from the Stack Space as roots.

The scavenger's operation is imperceptible to the user. To ensure that this is so, the scavenger will start to tenure objects from New Space and place them in Old Space if the number of survivors starts to slow down the speed of the scavenger's operation.

Incremental Garbage Collector

Unlike the scavenger, which only reclaims objects in New Space and Large Space, the *incremental garbage collector* (IGC) reclaims objects in Old Space, New Space and Large Space. It does so incrementally, recycling dead objects by placing their headers and their bodies on the appropriate threaded free list.



The IGC can be made to stop if any kind of interrupt occurs, or it can be made to ignore all interrupts. In addition, you can specify the amount of work that you want the IGC to perform, both in terms of the number of objects scanned or the number of bytes scanned — it will stop as soon as either condition is satisfied.

The IGC has five distinct phases of operation:

- Resting — the IGC is idle.
- Marking — the IGC is marking live objects.
- Nilling — the IGC is nilling the slots of WeakArrays whose referents have expired.

- Sweeping — the IGC is sweeping the object table, placing dead objects on the threaded free lists.
- Unmarking — the IGC is unmarking objects as a result of the mark phase being aborted, either at the user's request or because the IGC ran out of memory to hold its mark stack.

The typical order of operation is:

1. resting
2. marking
3. nilling
4. sweeping
5. resting

The unmarking phase is only entered if the mark phase is aborted, and it leaves the IGC in the resting phase when it is finished unmarking all objects. Each of the above phases is performed incrementally; that is, each can be interrupted without losing any of the work performed prior to the interruption. The IGC never performs more than one phase per invocation. This provision permits clients to specify different workloads and different interrupt policies for the different phases. Consequently, clients will need to wrap their calls to the IGC in a loop if they want it to complete all of the phases. There is protocol for doing this in the ObjectMemory class.

The object engine never invokes the IGC directly. Only Smalltalk code can run it. A typical memory policy might be to run the IGC in the idle loop, in low-space conditions, and periodically in order to keep up with the Old Space death rate. See the MemoryPolicy class for the default policy.

Compacting Garbage Collector

The compacting garbage collector is a mark-and-sweep garbage collector that compacts both object data and object headers. This garbage collector marks and sweeps all of the memory that is managed by the object engine except for Perm Space, whose objects are treated as roots for the purposes of this collector. This garbage collector is never invoked directly by the object engine, since the duration of its operation could be disruptive to the Smalltalk system.

Global Garbage Collector

The global garbage collector is a mark-and-sweep garbage collector that is identical to the compacting garbage collector except that it marks and sweeps all of the memory that is managed by the object engine, including Perm Space. This garbage collector is never invoked directly by the engine, since the duration of its operation could be disruptive to the Smalltalk system.

You might want to invoke the global garbage collector when you suspect that there are many garbage objects in Perm Space. This would reduce the size of the image file produced by a subsequent **File → Save As...**. It would also reclaim the space occupied by garbage objects in Old Space, New Space and Large Space that are only kept alive by references from garbage objects housed in Perm Space.

Since the global garbage collector uses the compiled code cache to store the mark stack, the default size may be inadequate for large images. If garbage collection fails due to inadequate memory, change the size of the compiled code cache at startup, by sending a sizesAtStartup: message to ObjectMemory. For details, see "Setting Object Engine Space Sizes" (below).

Data Compactor

The system also has an Old Space data compactor. Because this facility does not try to compact the object table, or mark live objects, it runs considerably faster than either of the two garbage collectors. It should be invoked when Old Space data is overly fragmented.

For details using the data compactor, see "Promoting Objects to Perm Space" (below).

Memory Policies

The object engine only supplies the low-level mechanisms for managing the object memory, allocation, and garbage collection. It is up to the Smalltalk memory-management code to utilize these mechanisms in a judicious manner. The latter belongs to the virtual image, and is accessible to the application developer.

The object engine provides an interface to the Smalltalk memory-management code via primitives, which is accessed via the classes ObjectMemory and MemoryPolicy. Developers wishing to access or modify the memory policies should use ObjectMemory's public protocol, or by creating a subclass of MemoryPolicy.

ObjectMemory

An instance of ObjectMemory represents a snapshot of object memory as it existed when that instance was created. The information contained in this object can be used to guide policy decisions for managing object memory (see the class MemoryPolicy for one such policy). This class also contains protocol for manipulating the state of object memory.

In general, if you want to access the current state of object memory, you would create an instance of this class and then send messages to the instance. If, on the other hand, you want to directly manipulate the state of object memory (for example, to grow object memory, to compact object memory, or to reclaim dead objects that exist in object memory), you would do so by sending a message directly to the class itself.

Because the information contained in this class is implementation dependent and because it may vary from release to release, it is recommended that this information only be accessed directly by the low-level system code that implements the various memory policies. Such policy objects should provide an adequate set of public messages that will permit high-level application code to influence memory policy without resorting to implementation-dependent code.

MemoryPolicy

Class MemoryPolicy implements the system's standard memory policy. This policy is composed of two regimes for the object memory: one for growth and one for reclamation. The growth regime is in force when memory usage is below the *growth regime upper bound*, the reclamation regime is in force whem memory usage is above it.

VisualWorks provides a default value for this upper bound, but it is the developer's responsibility to set it appropriately.

In the growth regime, only the scavenger and incremental collectors are active. The object memory is allowed to grow freely upon demand, up to the growth regime upper bound, at which point the reclamation regime is entered.

Under the reclamation regime, the object memory is not allowed to grow without a garbage collection of Old Space. Since garbage collection involves more overhead, overall performance is degraded somewhat during the reclamation regime.

> **Caution:** The default growth regime upper bound is 32 MB, *which is only suitable for small applications*. To avoid frequent garbage collections when using or deploying your application, you may need to adjust this setting. For details, see "Working with Memory Policies" on page 21.

When the reclamation regime is in effect, memory will be allowed to grow if garbage collection fails to make space available up to the *memory upper bound*. If the application attempts to grow memory above the upper bound the system will enter an *emergency low space condition*. The default MemoryPolicy response to an emergency low space condition is to interrupt the current active user process. For example, since Smalltalk stack frames are represented by Smalltalk objects, an infinite recursion can cause a low space condition, and the infinite recursion is interrupted once memory usage has grown up to the memory upper bound.

The memory upper bound is also adjustable via the Memory Policy settings (see "Working with Memory Policies" on page 21, for details). The default memory upper bound is 512 MB. On machines with less memory to devote to VisualWorks, this value is too high, and certainly too small for machines with more than 512 MB of memory (if you wish to use it for objects). Machines with less RAM than the VisualWorks upper bound, the operating system will typically begin paging furiously and thrash badly once memory usage exceeds the amount of RAM (the symptom being an error from the operating system that virtual memory has been exhausted). Consequently, if the memory upper bound is too high, an infinite recursion may take a long time to interrupt.

Unfortunately, on many platforms supported by VisualWorks there are no operating system APIs to discover how much free RAM is available. Hence VisualWorks does not automatically determine a suitable memory upper bound. Therefore *you* must choose a suitable memory upper bound for your installation. For single user development, we recommend

that you set the memory upper bound to be between 75% and 90% of total system RAM, and that you set the growth regime upper bound to be about 67% of the memory upper bound.

Free Space Upper Bound

MemoryPolicy grows Old Space and Fixed Space memory using the growOldSpaceBy: and growFixedSpaceBy: primitives, which, if they are implemented, employ the operating system's memory mapping facilities. Consequently, new segments can be returned to the operating system if the garbage collector can empty them. The compacting garbage collector compacts across segments and MemoryPolicy will release empty segments back to the operating system until the amount of free space is at or below the *free memory upper bound*. This upper bound determines how much free Old Space the system keeps in reserve for new allocations after a garbage collection. If your application cyclically allocates large amounts of memory and releases it only to allocate the memory once more, you may find it profitable to increase the free memory upper bound to reduce the amount of growth and shrinkage the system performs. For details on adjusting the free memory upper bound, see "Working with Memory Policies" on page 21.

By default, the free memory upper bound is 8 MB. This value is effective as long as it is significantly larger than the default growth increment, which is 1 MB. The default growth increment, which determines the minimum size of a newly allocated Old Space segment (its preferredGrowthIncrement), is adjustable in the initialization method MemoryPolicy>>setDefaults. Again, this is suitable only for small applications. Developers of applications with a memory footprint above 256 MB should consider raising this along with the free memory upper bound.

Default MemoryPolicy Behavior

MemoryPolicy objects are given the opportunity to take action during the following circumstances:

- During the idle loop
- When the system runs low on space

In addition, memory policy objects are responsible for determining precisely what constitutes a low-space condition.

An instance of MemoryPolicy takes the following actions in these circumstances:

idle-loop action
: Runs the incremental garbage collector inside the idle loop, provided that the system has been moderately active since the last idle-loop garbage collector. Lets the idle-loop garbage collector run until it is interrupted.

low-space action
: Responds to true low-space conditions. If the system is biased toward growth, then it attempts to grow object memory. If, however, it is not biased toward growth, or if object memory cannot be grown, then it tries various ways of reclaiming space. Failing that, it tries one last time to grow object memory. Failing that, it summons the low-space notifier.

  The most interesting of these steps is the reclamation step. An instance of this class will perform a full, compacting garbage collector only if the free entries in the object table are consuming a significant percent of Old Space. If, on the other hand, a compacting garbage collector is not needed, the policy object will try to reclaim space by simply finishing the incremental garbage collector (if one is currently in progress). If that doesn't free up enough space, then the incremental garbage collector is run from start to finish without interruption. Finally, a data compaction is performed if Old Space is sufficiently fragmented.

Managing the Object Memory

Several different mechanisms are provided to give application developers precise control over the object memory and the policies for managing it:

- Promoting Objects to Perm Space
- Setting Object Engine Space Sizes
- Working with Memory Policies
- Creating a Custom Memory Policy

Promoting Objects to Perm Space

Moving Old Space objects into Perm Space (and thus improving the efficiency of garbage reclamation) is done by creating an image by choosing **File → Perm Save As…** in the VisualWorks Launcher window.

Creating an image in this way is similar to making a snapshot except that all of the objects that are currently in Old Space will be promoted to Perm Space when the new image is loaded back into memory at startup time. For details on these spaces in the object memory, see "Perm Space" and "Old Space" (above).

Alternately, you can cause all of the objects in Perm Space to be loaded into Old Space at startup time if you create an image using **File → Perm Undo As…** in the VisualWorks Launcher window.

Note that the current state of object memory is not changed by creating a new image using **Perm Save** or **Perm Undo**. In other words, only the newly created image will contain a modified Perm Space. For example, if you use **File → Perm Save As…** to create an image and later in that same session you create a normal snapshot on top of that image, Perm Space is unaffected.

To place your application code in Perm Space, follow these steps before deploying an image containing the application:

1. Create an image using the **File → Perm Save As…** command. Then choose **File → Exit VisualWorks…** and start the new image. All of the objects that were formerly in Old Space will be loaded into Perm Space, including the application code.
2. A number of transient objects will also inhabit Perm Space, such as those needed to display windows on the screen — to remove them, perform a global garbage collection.
3. Create a normal snapshot.

4 To make subsequent loads on the same platform even faster, you may want to load the new image back into memory and perform one last snapshot.

This last step is useful because the global garbage collector compacts the objects in Perm Space, which forces the image loader to relocate these objects at startup time. By performing one extra snapshot, these objects will not need to be relocated on subsequent loads, when it is possible for the object engine to load them into their former locations.

Setting Object Engine Space Sizes

The default object engine memory space sizes are platform specific. Sizes for the following memory spaces can be adjusted at startup:

1. Eden (Object Creation Space)
2. Survivor Space
3. Large Space
4. Fixed Space
5. Stack Space
6. Compiled Code Cache
7. Old Space Headroom

To change any of these values, send sizesAtStartup: to the ObjectMemory class with an array specifying a multiplier for each space, then save and restart the image. The order of the array elements is as listed above.

Each multiplier must be a floating point $0 \geq x \geq 1000$. To get the requested memory size, the system applies the multiplier to the default size. A multiplier value of 1.0 yields the default size.

For example, to decrease Stack Space by 1/4 and increase New Space by 1/2, while leaving the others at default sizes, send the message:

```
ObjectMemory sizesAtStartup: #(1.5 1.0 1.0 0.75 1.0 1.0 1.0)
```

This sets the size for the image at next startup. To make the new sizes take effect, save the image, exit VisualWorks and then restart the image.

These values are recommended values only, and the object engine may start with a larger size if required to load the image.

Guidelines for Adjusting Memory Spaces

When adjusting memory spaces for most VisualWorks applications, you should first consider changing Eden (New Space):

All new objects are created in Eden. If this space is too small, then objects can get tenured in Old Space too quickly. On the other hand, if New Space is too big, the incremental collector that scans New Space can begin to impact performance. The "correct" size is generally a balance between processor speed and application behavior. Generally, the default setting is acceptable.

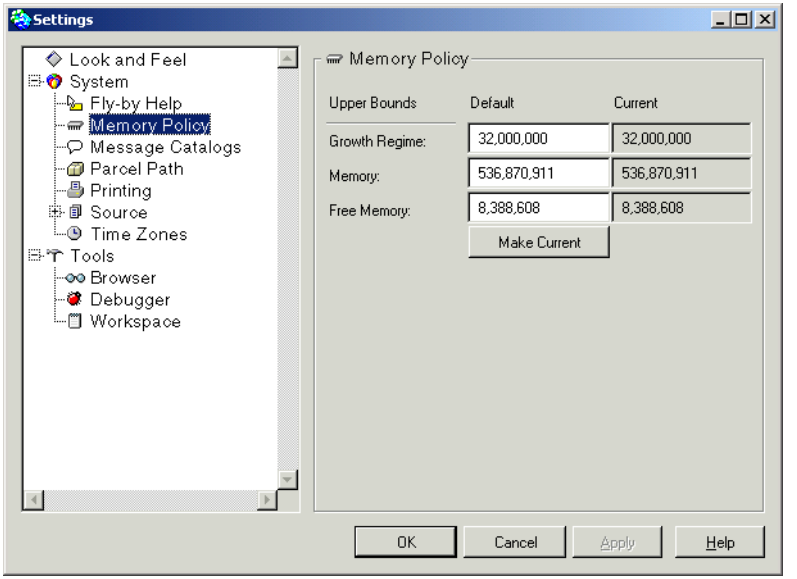
Since Old Space is grown automatically, there is no need to manipulate it using the sizesAtStartup: method. Instead, you may control the size of Old Space using the growth regime and memory upper bounds (for details on adjusting these values, see “Working with Memory Policies” on page 21).

Working with Memory Policies

Since the default MemoryPolicy is almost always sub-optimal for a particular application, it is considered good practice to adjust the policy before deployment. The following discussion explains the various parameters that are available, and offers some guidelines for adjusting them to maximuze the performance of your application.

For some applications, you may also want to create your own memory policy. For details, see "Creating a Custom Memory Policy" (below).

To examine the default MemoryPolicy settings, open the Settings Manager and select the **Memory Policy** page (choose **System → Settings** in the VisualWorks Launcher window):



The default policy allows the object heap to grow unrestricted up to the **Growth Regime** upper bound. Attempting to grow the object memory beyond this bound will invoke the garbage collector to reclaim memory.

The default policy does not allow memory to grow above the **Memory** upper bound. Instead, the currently active process is interrupted with a low-space condition.

The **Free Memory** upper bound specifies the amount of memory that is always dedicated to VisualWorks. After a garbage collection, the default memory policy attempts to return any free memory above this bound to the operating system.

Guidelines for Adjusting the Basic Settings

While the default MemoryPolicy is generally suitable for application development, it can almost always be optimized for deployment. The **Growth Regime** upper bound, for example, is almost always too large for small applications, and too small for large applications.

Especially for large applications, the **Memory** upper bound is almost always too small. For an application that uses more than a gigabyte of RAM, it is *absolutely* necessary to adjust the MemoryPolicy defaults.

The best way to set these parameters is to first measure your application's memory usage. To get an idea of memory usage, use:

```
ObjectMemory current dynamicallyAllocatedFootprint
```

The number returned is the total size (in bytes) of the dynamically allocated footprint.

When measuring memory usage, it's important to measure the size of the object memory at points of high load. Insert test code in the application to take measurements (perhaps using a log file), and if you can identify a point of high load, get the system to perform a garbage collect, making sure to record the dynamicallyAllocatedFootprint both before and after the garbage collection. If measurements show that no memory is reclaimed and the application is still functioning, then the growth regime upper bound is too small.

Use the results to set the MemoryPolicy parameters. The first and most obvious adjustments can be made to the **Growth Regime** and **Memory** upper bounds. The **Growth Regime** upper bound should be at or below the peak usage. This bound must accomodate the maximum expected working set for your application.

The **Growth Regime** upper bound should be set to keep the time spent in garbage collection to an acceptably low interval. Attempts to grow the object memory beyond this limit cause VisualWorks to garbage collect before asking for more heap memory from the host OS. If the limit is too low, the application spends too much time garbage collecting. If it's too high, the application wastes memory by not garbage collecting.

The **Memory** upper bound should be set to the application's maximum footprint plus a safety margin. You should keep in mind that since memory growth is not allowed beyond the upper bound, computations will be interrupted with a low space error if memory usage ever reaches the upper bound. The safety margin should accommodate this error.

MemoryPolicy has an availableSpaceSafetyMargin instance variable that defines the amount of free space to maintain as a safety margin, ensuring that the low space actions can be performed. By default, this instance variable is intialized to the sum of ObjectMemory>>stackZoneFlushBytes and ObjectMemory>>emergencyDebuggingHeadroom. This suits the default low space actions. Your application may require a different value (if, for example, its response to an emergency low space condition would be to log an error to a log file or across a socket).

The **Free Memory** limit may need to be raised to a higher number, so that the application doesn't release as much free memory after a garbage collection. While in principle it sounds good to have the application return memory to the host operating system after garbage collection, in practice this may degrade performance.

The VisualWorks MemoryPolicy contains a number of other adjustable parameters, which are not shown in the Settings Manager. To adjust these, you must create your own custom policy class.

Creating a Custom Memory Policy

For many applications, you can improve performance by using a custom memory policy. You may can any aspect of the policy, inducling both the constants and algorithms used to manage the object memory.

The basic steps to create a new policy are:

1. Define a new subclass of MemoryPolicy.
2. Make sure to implement the setDefaults method, invoking the version in the superclass, and then setting the appropriate variables. E.g.:

```
setDefaults
    super setDefaults.
    preferredGrowthIncrement := 10000000.
```

3. Install the new policy:

```
ObjectMemory installMemoryPolicy: MyMemoryPolicy new setDefaults
```

In this example, we have created a new policy that uses a different growth increment. This is the amount by which the object memory is grown when an allocation failure occurs during the growth regime.

By default the growth increment is 1,000,000 bytes, which means heap segments will be at least that big. In an application using more than a gigabyte of memory, this yields an awful lot of small segments. Setting the growth increment to something like 1/100 of available RAM might be better.

For a complete description of the MemoryPolicy API, see its class comment.

MemoryPolicy Strategies

A custom memory policy might perform application-specific actions, flushing application caches, making polciy decisions about process allocation, and so forth. For example, the VisualWorks Application Server uses VisualWave.ServerMemoryPolicy to assess system load, expire web existing sessions, and refuse new connections.

Another example of a custom policy would be one that maintains a pre-specified range of memory usage (suggested by Alex Pikovsky). That is, if the available memory drops below a lower threshold, the policy enters the growth regime, and if it reaches an upper threshold, it enters the reclamation regime.

For the purposes of the example, let's say the lower threshold is 10Mb, and the upper threshold is 50Mb. We can implement this in a subclass of MemoryPolicy that contains the two following methods:

setDefaults

```
super setDefaults.
maxMemorySize :=  Core.SmallInteger maxVal * 4.
self
  memoryUpperBound: maxMemorySize;
  preferredGrowthIncrement: 40000000; "40 MB"
  growthRetryDecrement: 1000000; "1 MB"
  maxHardLowSpaceLimit: 5000000; "5 MB"
  availableSpaceSafetyMargin: 2500000; "2.5 MB"
  contiguousSpaceSafetyMargin: 1000000; "1 MB"
  threadedDataIncrement: 1000000; "1 MB"
  freeMemoryUpperBound:  50000000; "50 MB"
  growIfFreeBytesLessThan: 10000000. "10 MB"
```

And:

favorGrowthOverReclamation

```
"Answer true if we want to react (at this point in time) to the low-space
condition by growing memory rather than reclaiming memory."

^self memoryStatus availableFreeBytes <= self growIfFreeBytesLessThan
```

Preparing for Deployment

The following steps are recommended for deploying a VisualWorks image:

1. Load application code.
2. Prepare to create a new image, promoting Old Space objects into Perm Space (in the Launcher window, select **File → Perm Save As...**).
3. Run the Global Garbage Collector (in the Launcher, select **System → Collect All Garbage**).
4. Create a snapshot (in the Launcher, select **File → Save As...**).
5. Set MemoryPolicy paramaters correctly (for details, see "Working with Memory Policies", above).
6. Run the Global Garbage Collector (in the Launcher, select **System → Collect All Garbage**).
7. Save the image in a ready-to-run state.

Implementation Limits

The following table gives the size limitations for various aspects of the VisualWorks system. A limit of “None” implies that no hard limit exists, though available address space (32 bits) is an upper bound in every case.

Unit	Limit	Comment
Number of objects	None	Objects are limited only by address space. Average object size is on the order of 64 bytes, so the maximum number of objects available is approximately 2^{32} / 64 = 67108864 when 4 GB of memory is available.
Object size	256 MB for byte objects	2^{28} bytes for byte objects
	1 GB for pointer objects	2^{28} slots for pointer objects
Named instance variables	256 per class	Includes inherited instance variables
Method variables	255	Includes arguments, named temporary variables, unnamed temporary variables (needed to implement to:do: loops, etc.). Also includes pushes and pops, so the effective limit may be a little less.
Block variables	255	Includes block arguments and temporaries; in some circumstances, it also includes arguments and temporaries from outer scopes to which the block refers. Also includes pushes and pops (see above).
Method literals	256	Includes ordinary literals (strings, numbers, etc.), message selectors (other than about 200 of the most common selectors), static variables (global, pool and class) that are referenced, and one for each block.
Block nesting	256 levels	
Method branches	1023 bytes, forward or backward	This does not limit the length of regular code. In practice, it means that the body of an open-compiled loop or conditional cannot be longer than 1023 bytes.