

Space Management for O-O Programming

Alex Kodat
Sirius Software Inc.



Sirius Software, Inc.

Model 204 Space Management

- Two main areas to store data
 - Servers – User/thread specific data only
 - Allocated in arbitrarily small chunks
 - Extremely efficient access path
 - Files – Shared among all threads
 - One pool shared among all threads
 - Allocated in page-size (6184-byte) chunks
 - Individual pages can be thread-specific or shared
 - Somewhat more expensive access path
- Other odd areas to hold control data
 - Record locking table
 - Miscellaneous areas too small to worry about



Application Data

- **Always** accessed from main storage/memory/core
- Can be moved to/from main storage from/to disk
 - Disk itself can be/have its own main storage
 - In which case, disk subsystem moves data between its main storage and CPU and between its main storage and disk
- Can also be moved to other external media
 - Tape, CD, etc..
 - Not practical for most online applications



The Primary Goal of Storage Management

- Fitting 10 pounds of manure into a 5 pound bag
 - Military term is *blivet*
- Managing more data than available real memory for CPU
- Managing more data than available with CPU architecture
 - 24-bit storage (16M) in the old days
 - 31-bit storage (2G) now



The Secondary Goal of Storage Management

- Minimizing CPU cost of managing storage
- Don't want CPU overhead to go up as more main storage is made available to or used by applications
 - Avoid sequential scans of large data structures
 - So adding main storage is a pure win rather than a trade-off



Persistent/Historical Data

- **Must** be stored on persistent medium
 - Usually means disk for online applications
 - Also means regular backups to tape or redundant disk
- Quantity of data grows over time
- Quantity of data grows with application sophistication
- Expands to fit disk available to it
 - If the space is there, why not use it?
- So generally outstrips available main storage by a lot
 - And growth matches or exceeds main storage growth



Short-term Application Data

- Can include copies of persistent data
- Doesn't need to be stored on persistent medium
- Quantity of data grows with application sophistication
 - So requires work to increase
- Quantity sometimes grows to provide algorithmic simplifications
 - Throwing memory at a problem
- So generally outstrips available main storage
 - But growth generally slower than main storage growth
 - Ultimately, all short-term application data will be in main storage
 - We're probably already there except for the architectural (2G limit) issue



The Classic Database Trade-off

- Main storage for persistent data vs. short-term application data
 - Insatiable appetite for storage of persistent data
 - High penalty for slow access to short-term data
- Because main storage now exceeds most people's short-term application data requirements, this trade-off now largely doesn't exist
 - Except for managing 31-bit storage
 - Short-term data gets as much as it needs, all the rest goes to persistent data
 - Partially because of 204's small per-user main storage “footprint”



Organizing Short-term Data in 204

- Most short-term data sits in the server
- One server per logged-in user
- Running users must have server in virtual storage server area
 - Area **should** always be in real storage
 - Number of servers can exceed number of server areas
 - 204 swaps servers into and out of those areas as needed
 - Swapping can simply be to/from 31-bit virtual storage from/to 64-bit storage



Servers as the Main Unit of Short-term Storage

- Extremely efficient access to data once in server
- All of a given thread's data contiguous so no fragmentation of user data
- Storage management in large (server-size) blocks so no system-wide fragmentation
- Data movement done in large (server-size) chunks, so very efficient (vs. individual page-faults)
 - Ultimately data movement will go away when servers sit in 64-bit storage
- Isolation of most per-user data dramatically reduces chance of memory *leaks*
 - And limits the damage in the case of application/system errors




Traditional Storage Use by 3GLs

- One approach - every variable has a static area associated with it
 - The approach used by 204
- Another approach - every variable has an area in the current stack frame associated with it
 - Space allocated when subroutine/procedure/method called and returned when subroutine/procedure/method returns
- So space usage fairly tidy

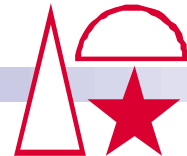
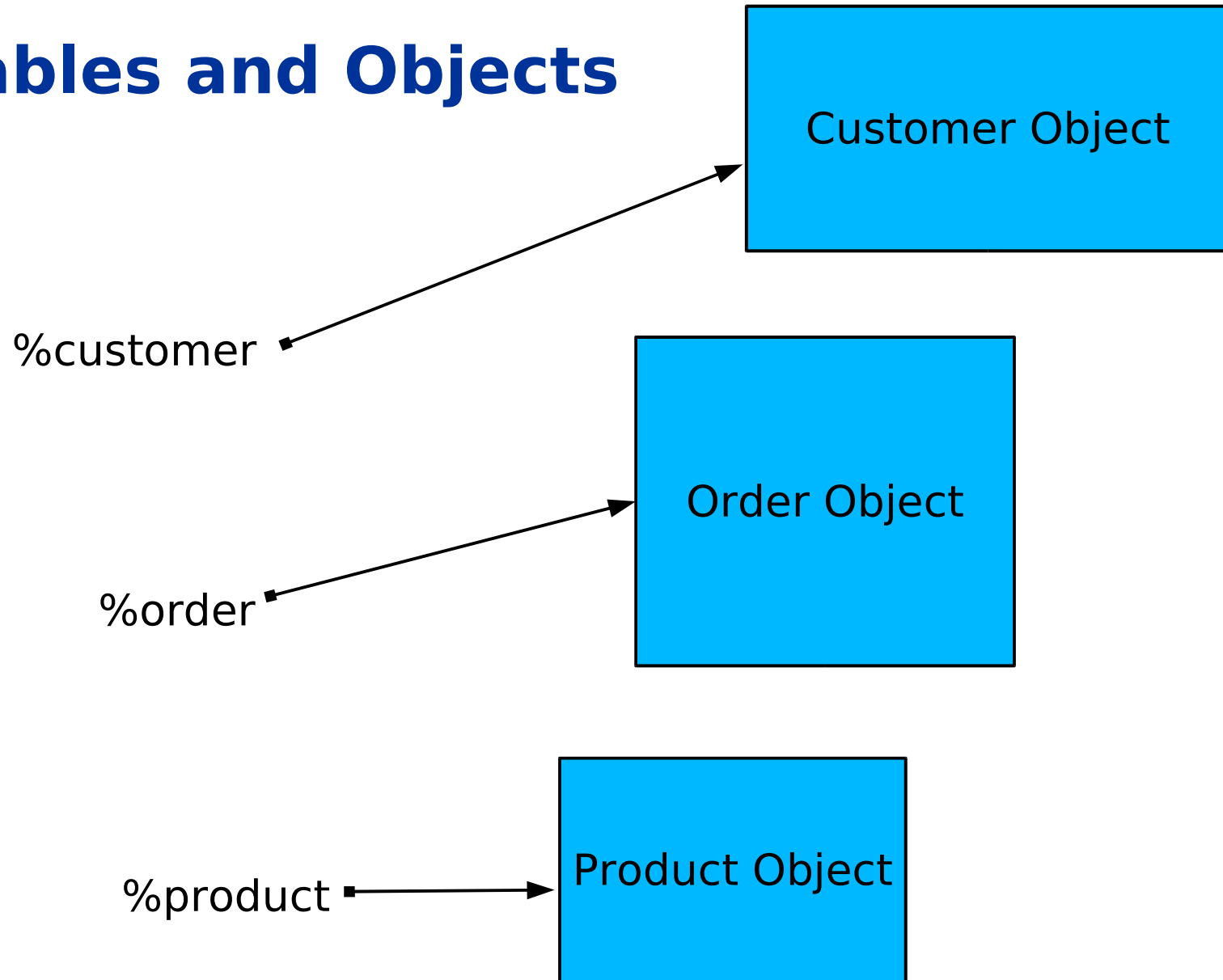


Storage Use by O-O Languages

- Same as traditional 3GL for object variables
 - But object variables are **not** the actual objects
- Object space allocation is dynamic
 - Space allocated for an object when object is *instantiated*
- Objects can be quite large and objects of different classes have different sizes
 - Because different classes have different numbers and types of variables
 - Objects are more or less collections of related variables 
- There can be arbitrary numbers of objects of any class



Variables and Objects



Let's See That in User Language

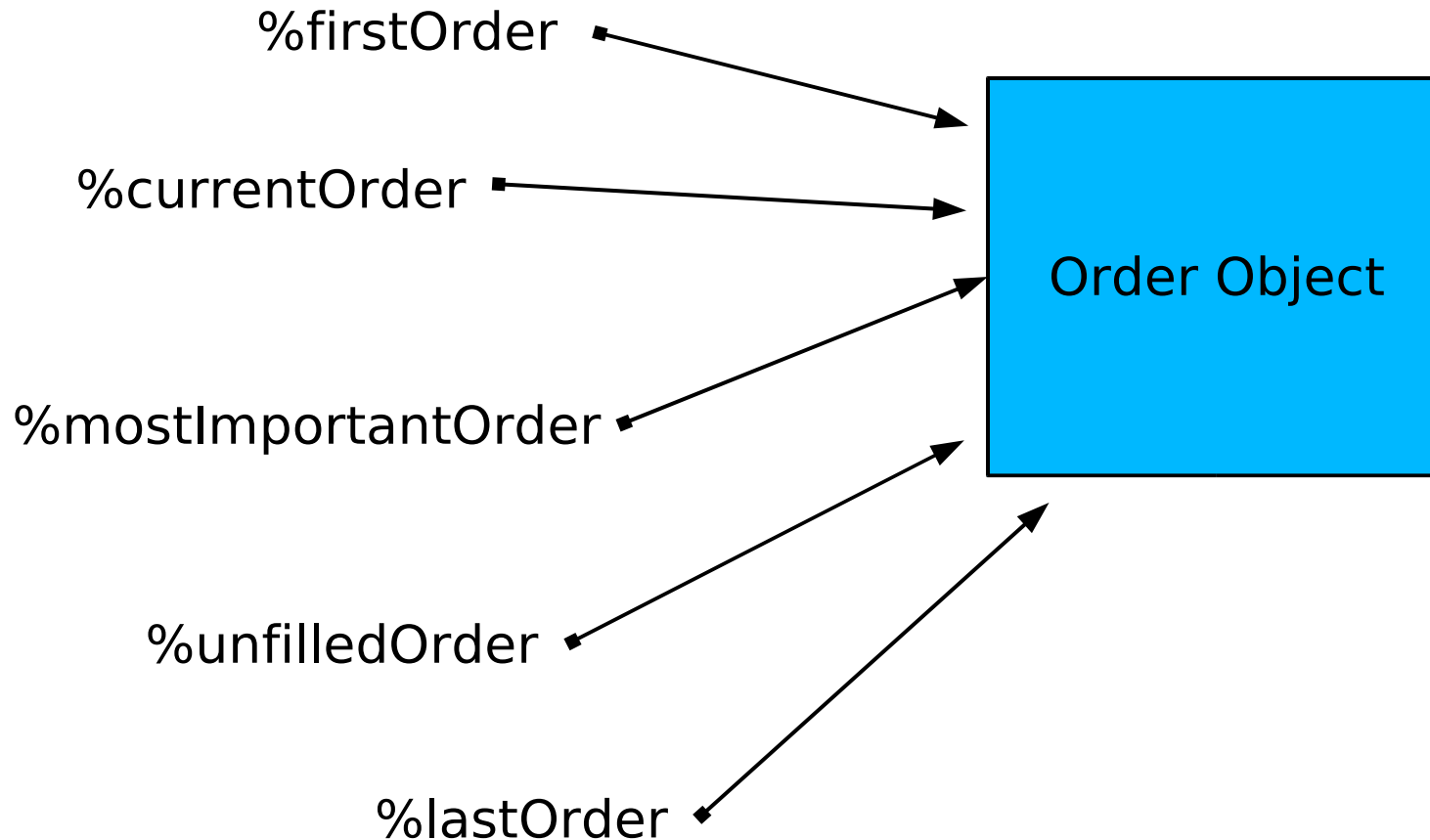
```
%customer      is object customer
%product       is object product
%order         is object order
...
* Let's create a new customer object and have %customer point to it
%customer = new
...

* Let's create a new product object and have %customer point to it
%product = new
...

* Let's create a new order object and have %order point to it
%order = new
...
```



Multiple Variables Can Reference (Point to) the Same Object



And in User Language

```
%firstOrder      is object order  
%currentOrder    is object order  
%mostImportantOrder is object order  
%unfilledOrder  is object order  
%lastOrder       is object order
```

```
...  
* Let's create a new order object and have %order point to it
```

```
%order = new
```

```
* And for now lets set all other order variables to point to it
```

```
%firstOrder      = %order  
%currentOrder    = %order  
%mostImportantOrder = %order  
%unfilledOrder  = %order  
%lastOrder       = %order
```

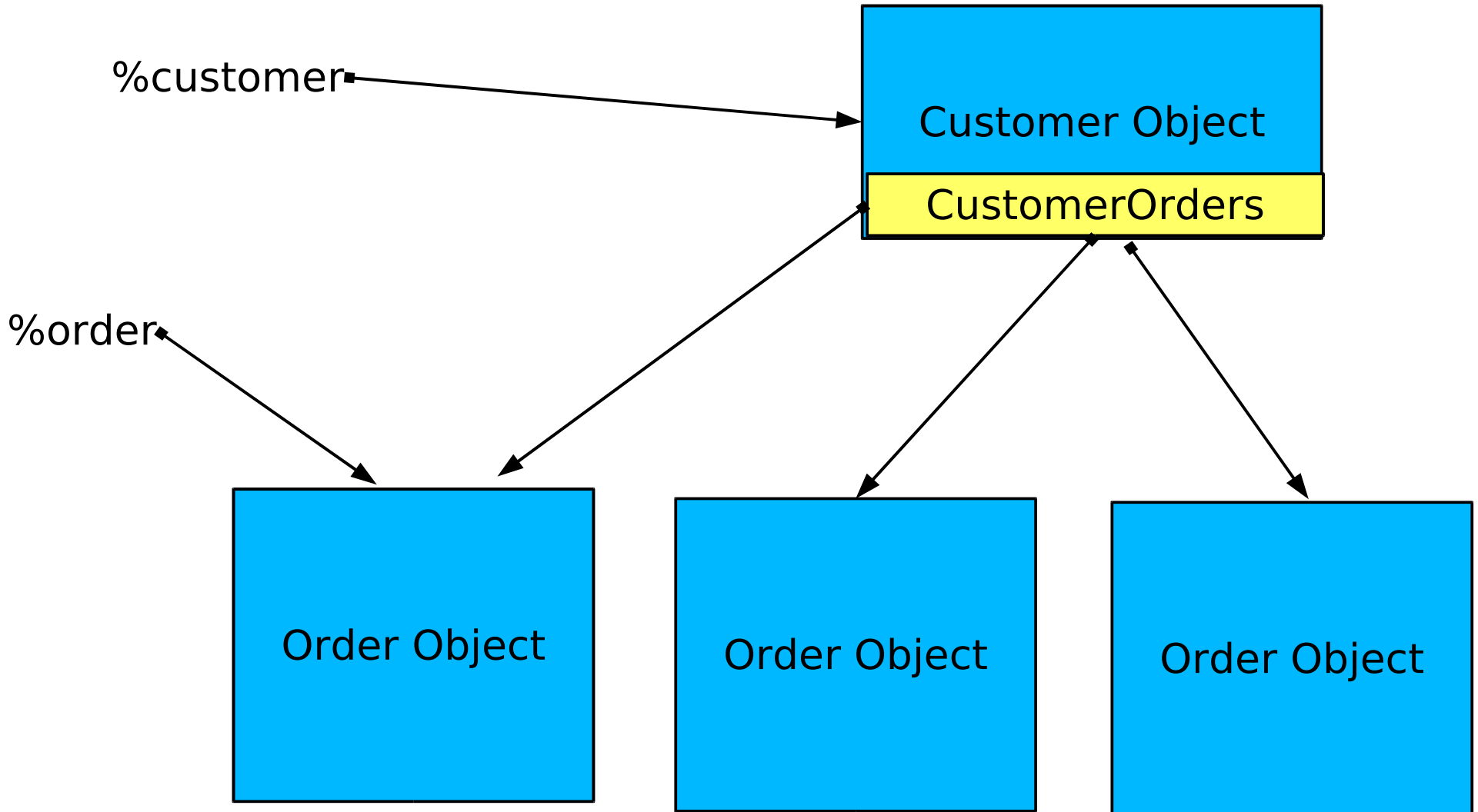


Why Would We Have Multiple References to the Same Object?

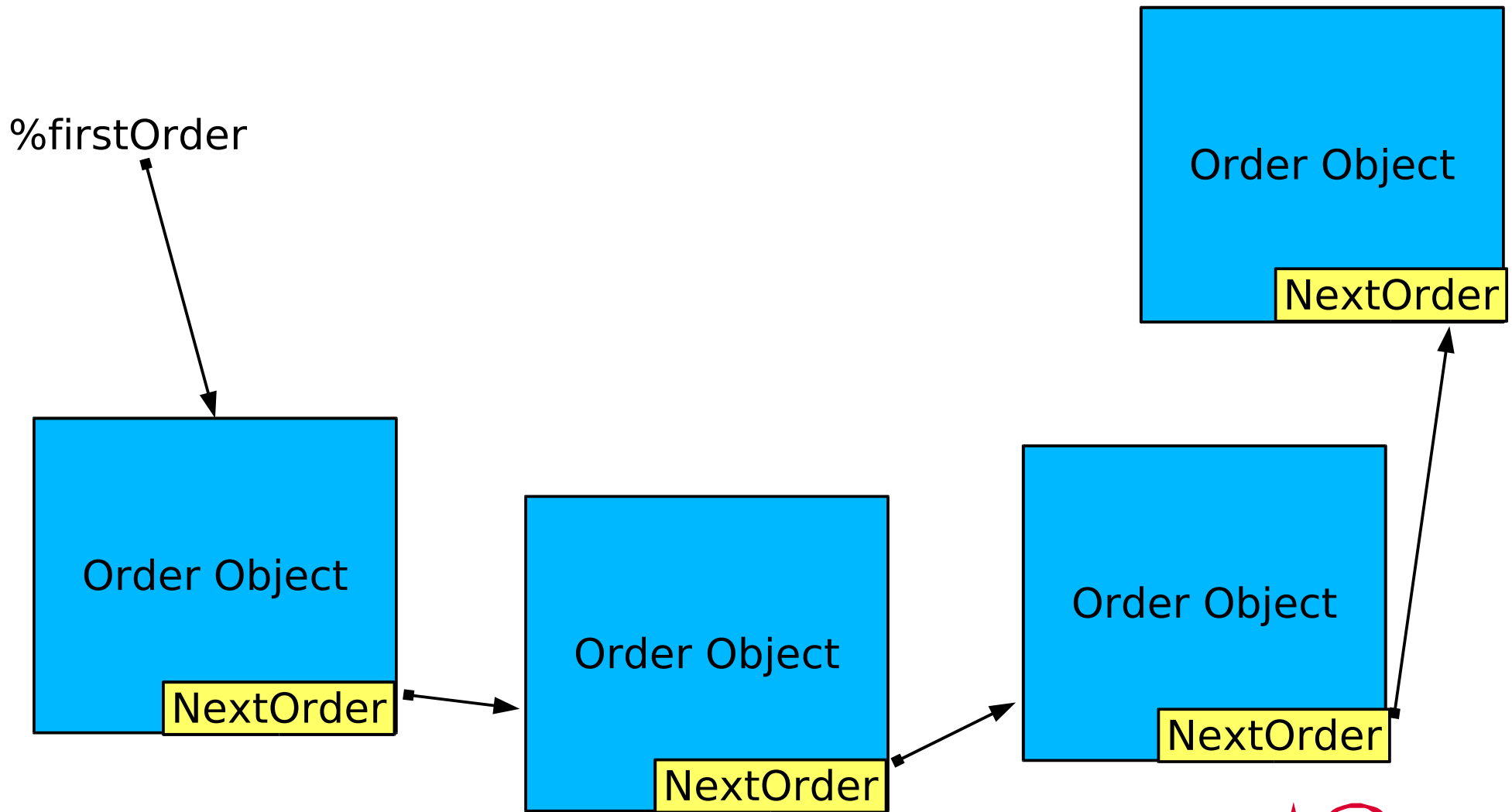
- Because it's useful
 - A given object (order) might be referenced in different ways in the same program (first, last, current, etc..)
- Why not copy the objects?
 - Because that's a formula for a total mess
 - Update of one copy not reflected in others
 - Attempts to correct this are a nightmare
 - It's inefficient



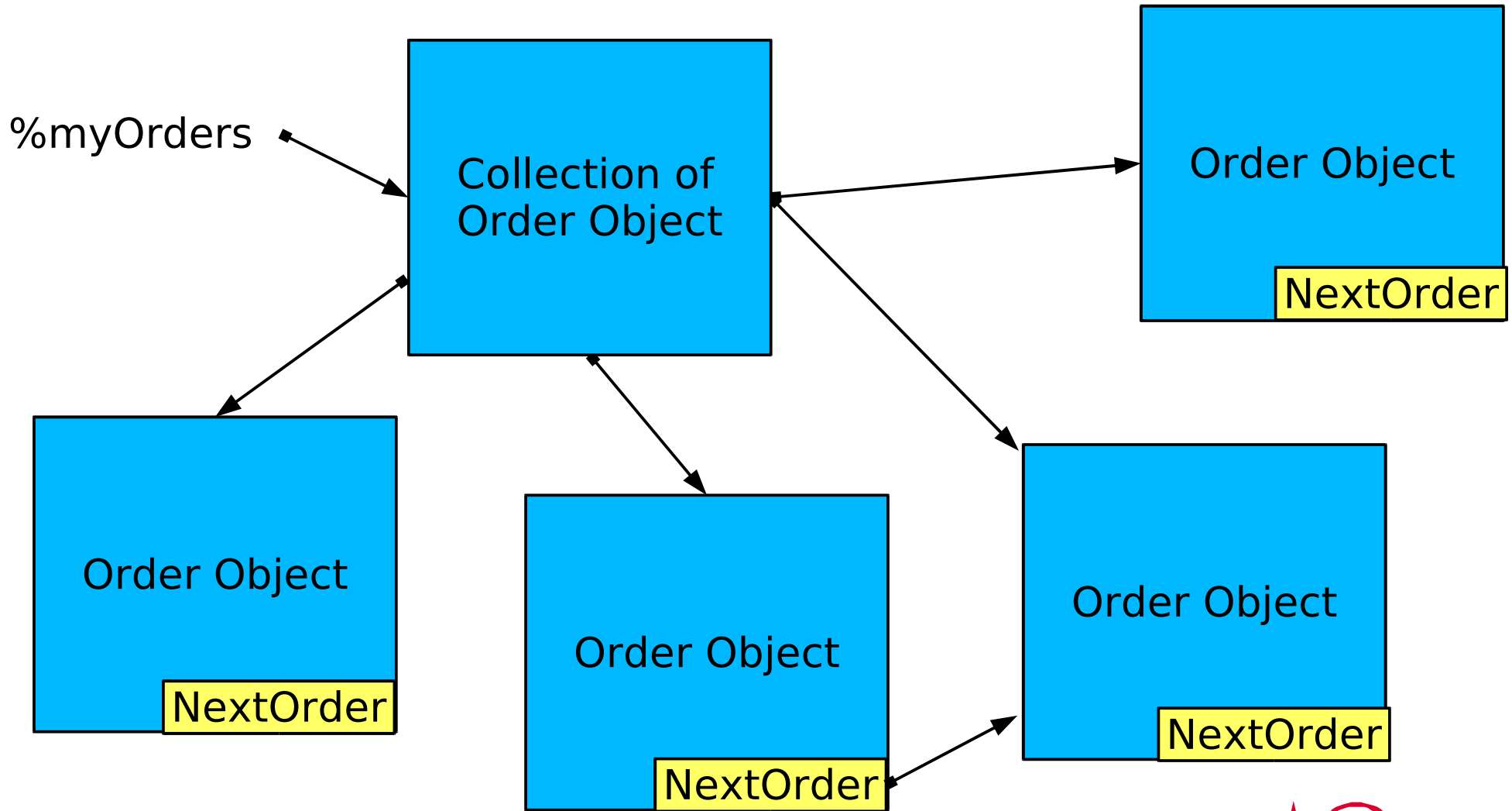
Object Can Also Reference Each Other



So You Can Have Chains of Objects



And Collections of Objects (Which are Themselves Objects)



So How Many Objects Can Be Created in a Program?

- Largely impossible to tell
- Can be fewer than number of object variables
- Can be (many) more than number of object variables
- So how can space be allocated for objects?
 - It must be done dynamically at run-time
 - Have a pool of space (sometimes call a *heap*) and when a new object is instantiated, space is taken from the heap for the object
 - When the program is done with the object, the space is made available again



Reclaiming Space for Unused Objects

- Occurs when program indicates it's done with object
 - By using Destroy, Nuke, Discard (Janus SOAP ULI) methods
 - But this is a hassle for the programmer
 - And mistakes can cause unreclaimed space, i.e. memory leaks
- Best if the environment can detect an object is no longer needed

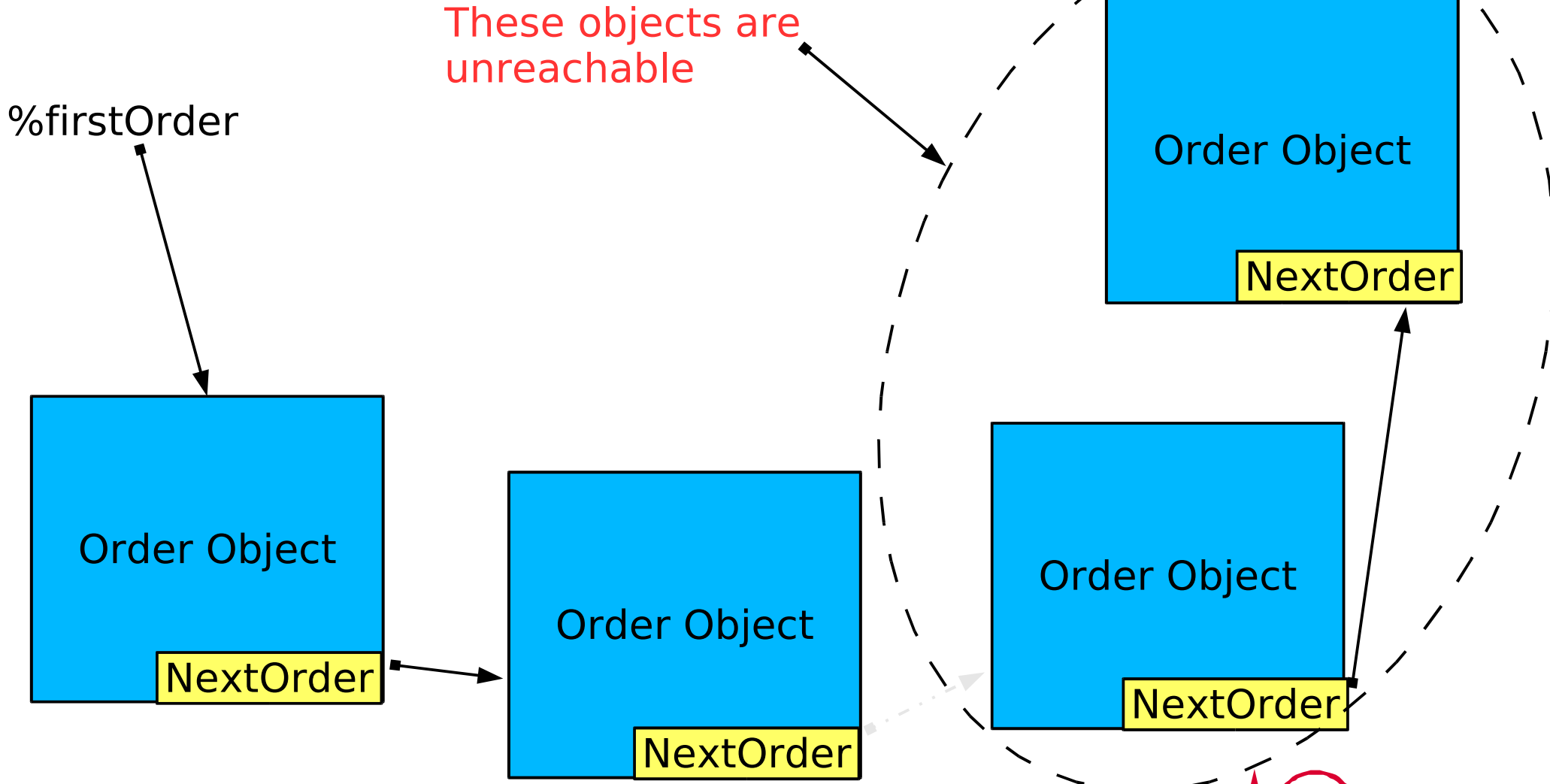


Detecting An Object is no Longer Needed

- An object is no longer needed if there is no way to get to it by following pointers starting at an object variable
 - Think about it
- Two major techniques for detecting this
 - Reference counts
 - Garbage collection



Unreachable Objects



Reference Counting

- Every time a variable is set to point to an object, that object's reference count goes up by one
 - Including variables inside objects
 - `%x = new` creates a new object and sets its reference count to one (`%x` references it)
- Every time a reference to an object is removed (by assigning its former referring variable to another object or null) the object's reference count goes down by one
 - If the count goes to zero, there are no more references to the object so the object is unreachable and its space can be reclaimed



Reference Counting in User Language

```
%firstOrder      is object order  
%currentOrder    is object order
```

```
...  
* Let's create a new order object and have %order point to it.  
* The new object's reference count is now 1.
```

```
%order = new
```

```
* Now we assign another variable to reference the same object.  
* The object's reference count is now 2.
```

```
%firstOrder      = %order
```

```
* Now we create another order object and have %order point to it.  
* Since %order no longer points to our original object its new  
* reference count is now 1.
```

```
%order = new
```

```
* Now we set %firstOrder to null. Since it no longer points to our  
* original object its new reference count is now 0. Since the  
* reference count is now 0, the object can be discarded.
```

```
%firstOrder = null
```



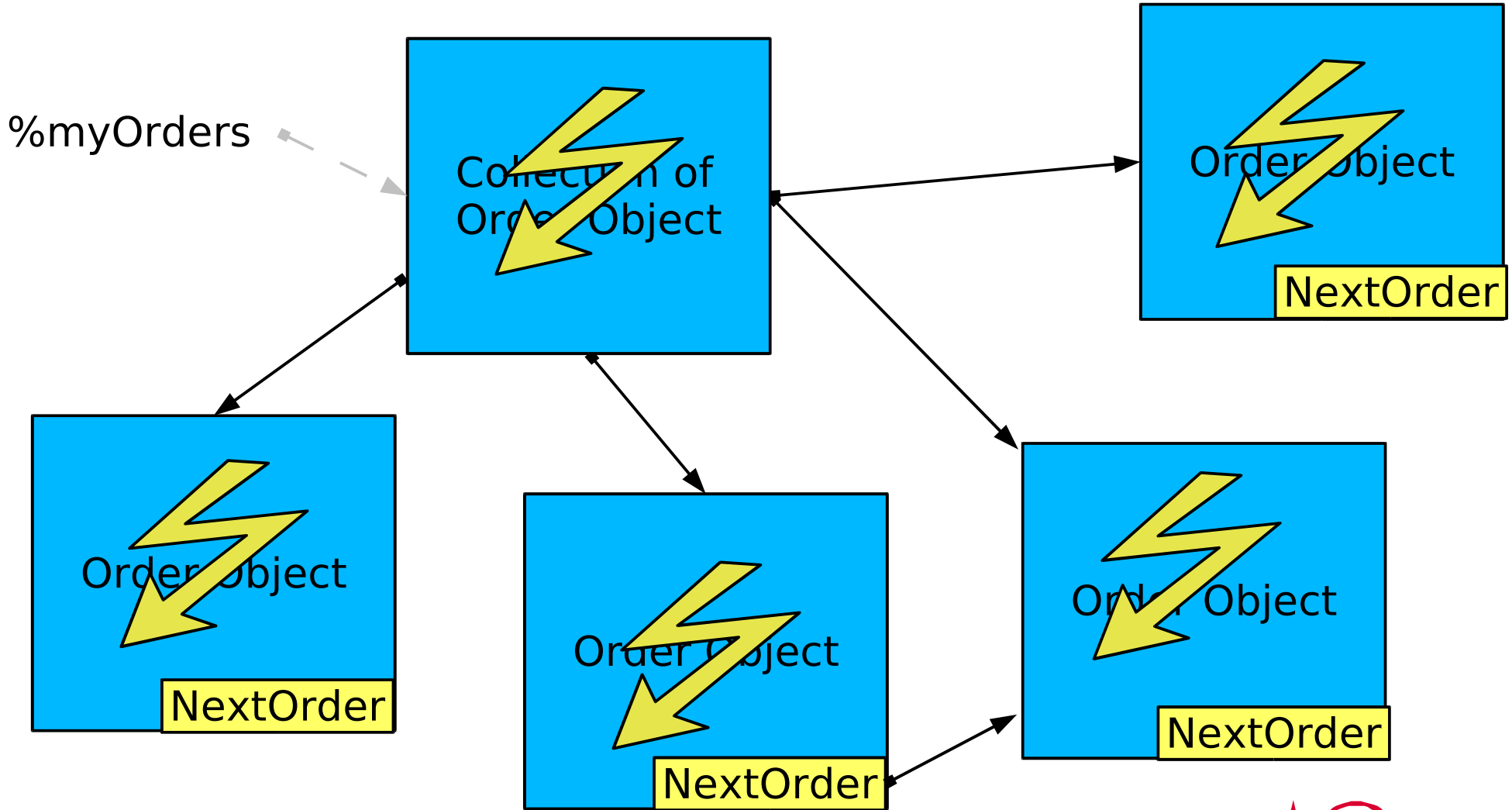
Cascading Discards

- If an object is explicitly or implicitly (use count goes to 0) discarded all **its** references to other objects are eliminated
- This decrements the reference counts of some other objects
- Some of those reference counts might go to zero
- So those objects are discarded
- And so on

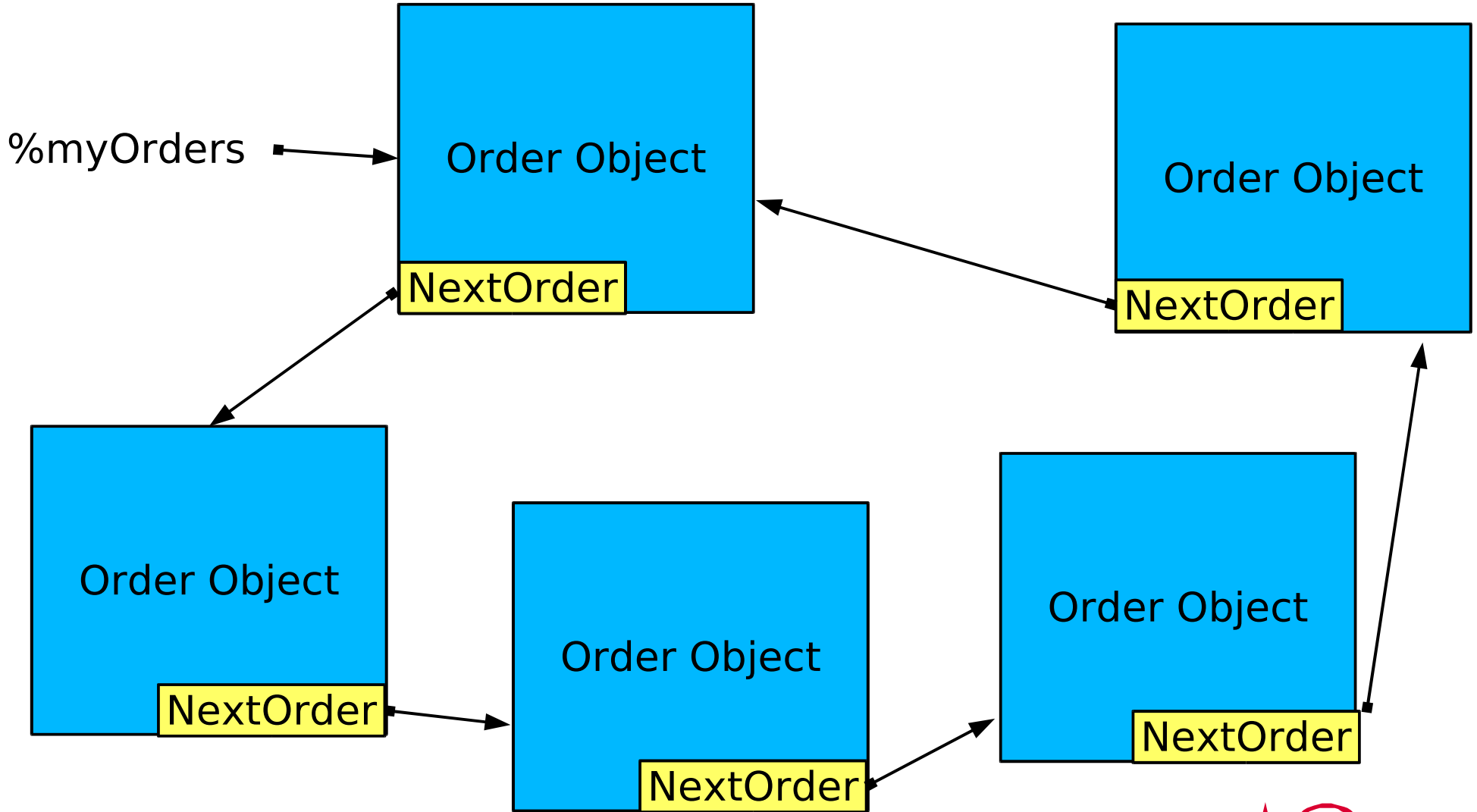


Cascading Discards

When %myOrders set to null, the collection is discarded and then all the orders it points to



Quiz: What is each object's reference count? What is it if %myOrders is set to null?



Object Cycles or Rings (was this what Wagner was thinking of?)

- The Achilles Heel of reference counting
 - Since each object in cycle has one reference to the next, the reference counts can never go to zero, even if the ring is unreachable
- Extremely difficult to detect cycles in the general case
- While cycles of objects all in the same class is most common, heterogeneous cycles are possible
- “In the know” applications can do an appropriate explicit discard to break the ring and prevent the problem



Garbage Collection

- The alternative to reference counting for automatic object cleanup
 - AFAIK, these are the only two choices



How Does Garbage Collection Work?

- 1 Scan all objects and mark them as unreachable
 - 1 Code that allocates objects must make it possible to find them independent of references
- 2 For each object variable check the object it references
- 3 If object already reachable, back to step 2 for next variable
- 4 Mark object reachable
- 5 For each object reference in object in step 4 check the object it references
- 6 If object already reachable, back to step 5 for next variable
- 7 Otherwise to step 4 for referenced object



Garbage Collection Pass 2

- After first pass, all objects that are marked unreachable are, well, unreachable
- So scan all objects one more time
- Reclaim space associated with any unreachable objects
 - I.e. Discard unreachable objects



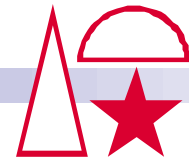
The Advantages of Garbage Collection

- Not fooled by cycles
 - It's a proof of the pudding algorithm
- It visits one object per reference
 - So largely linear cost with respect to number of objects
 - No N^2 behavior
- Can also move objects to reduce storage fragmentation in pass 2.
 - Must fix up references then, too
 - This can be complicated and expensive
 - But then, storage fragmentation can get expensive



The Disadvantages of Garbage Collection

- Big linear scans of all objects can be expensive
 - Can eat 10% or more of CPU in a typical system
 - On the other hand, who cares about 10%?
 - Maybe you'd burn that a little bit at a time, anyway, by maintaining reference counts
- Requires significant overallocation of space
 - Since cleanups delayed to minimize overhead
 - Often increases storage requirements by factor of two or more over real storage usage
- So requires weighing/trading-off CPU overhead vs. wasted storage



Garbage Collection or Reference Counting?

- The industry standard is garbage collection
- Much easier to code
 - One size fits all
- Handles the bad case of cycles
- Who cares about burning CPU/storage?
 - Especially storage?
 - And CPU waste is relatively small (< 20%, typically)
 - Especially if it makes programmers lives easier



The Janus SOAP ULI Storage Management Approach (cont...)

- Allocates a fixed number of slots in VTBL and STBL for each class of objects in a request
 - Allocation is fixed at compile-time
- Objects must be in one of the pre-allocated slots when being used
 - Swapped to/from CCATEMP if more objects than object slots
 - Kind of like FSCB swapping for screens and images
 - Except more than one slot available per class
- Object reference counts maintained.
 - When the count goes to zero (or object explicitly discarded), slot in VTBL/STBL marked available and/or space in CCATEMP reclaimed (sort of)



The Janus SOAP ULI Storage Management Approach (...cont)

- Swapped out data maintained in CCATEMP on an internal \$list (natch)
 - So swap \$list can get fragmented
- Explicit GarbageCollect call provided to clean up cycles
- If there are no global/session objects, all objects discarded at end of request
 - So the presence of a single global/session variable changes object cleanup behavior at end of request



Advantages Of Janus SOAP ULI Storage Management Approach

- Storage isolation of objects by thread (server) dramatically reduces storage fragmentation issues
- Pre-allocating space for each class reduces storage fragmentation
 - Also makes it possible to move objects between slots without having to pre-emptively fix up references
- Small storage footprints for “well-behaved” applications (applications that have relatively few objects of a class at a time)
 - Pathological requests simply do a lot of object swapping
- No expensive garbage collection required if no cycles or cycles cleaned up in UL program



Disadvantages Of Janus SOAP ULI Storage Management Approach

- Pre-allocating space for each class could waste space
 - Because space might be allocated that's never used
 - Space can't be shared among objects of different classes
- Does not lend itself to objects shared among threads
 - Though shared system objects **high** on our punch list
- Sequentially processing large numbers of objects can be expensive
 - One swap-in/out per object processed
 - Swap-in/out costs can dominate processing
 - Though swapping is pretty efficient



So What Should Programmers Worry About for Storage Management for Janus SOAP ULI?

- Object cycles
 - Avoid them
 - Clean them up manually if unavoidable
 - Cheaper than garbage collection
 - Throw in `%(object):garbageCollect` calls at appropriate places if too hard to manually clean up cycles
- Object swapping
 - More heavily accessed objects in a class than slots available for them



Detecting Too Much Object Swapping

- First, look at system OBJSWAP stat
 - Is it scary? Is so, keep investigating
- Look at since-last OBJSWAP stats
 - Look for requests with scary numbers
 - SirAud needs to be fixed to summarize this (sigh)
- Use OBJSTAT user parameter to get details about server table usage and swapping for a request.
 - Output can go to screen or audit trail



OBJSTAT Output

By class and total

```
MSIR.0884: OBJECT SYSTEM:DAEMON: objects/VTBL/STBL - 2/6/0, count/pages swapped 0/0
MSIR.0884: OBJECT SYSTEM:STRINGLIST: objects/VTBL/STBL - 3/18/0, count/pages swapped 210/210
MSIR.0884: COLLECTION SYSTEM:NAMEDARRAYLIST OF STRING LEN 255: objects/VTBL/STBL - 2/15/1536, -
count/pages swapped 0/0
MSIR.0884: *TOTAL*: objects/VTBL/STBL - 7/39/1536, count/pages swapped 210/210
```

Indicates space allocated (always same for compile/evaluation):

Objects:	Number of slots allocated
VTBL:	VTBL space allocated in VTBL units (32-bytes)
STBL:	STBL space allocated in STBL units (bytes)

Indicates swapping counts

Count:	Number of swapped
Pages:	Number of 204 pages (6124 bytes) swapped



Setting OBJSTAT in Production

- Required to catch applications that **occasionally** go crazy object swapping only in production (natch)
- Set OBJSTAT to X'02' to have post evaluation stats go to journal
- But, this will produce a **flood** of junk to the journal
 - Since stats will be produced for every class in every procedure, whether or not it swapped at all
- So, also set OBJSTMIN
 - Minimum number of swaps required for a class in a request before object swap stats displayed (sent to journal) for that class
 - 100 might be a reasonable first cut at this



What to Do About High Object Swap Counts

- Change application to reduce bouncing around between objects of a class
- Use Sirius MinObjects compiler directive to increase number of slots allocated for a class
 - *Sirius MinObjects Order 10*
 - If this number \geq max number of objects ever referenced in a request, all object swapping will be eliminated
 - Of course, overallocation wastes server space
- Shrug
 - Swaps might be unavoidable
 - If request scan 1,000 object sequentially 10 times, 10,000 swaps is probably unavoidable



A Confession

- Currently Janus SOAP ULI allocates a minimum of two slots per UL class and three slots per system class
 - Even if no more than one object is ever instantiated!
 - This simplifies complex cases where two objects of the same class are referenced in same statement
- So very wasteful for large objects with at most a single instance in a request
- We **will** fix this... **soon**



Summary

- Janus SOAP ULI storage management strategy will produce applications with small storage footprints (servers) and not much storage management CPU overhead in most “normal” applications
- Beware object cycles
 - Use GarbageCollect as needed
 - Automatic garbage collection (on as possibly needed basis) coming soon
- Check those OBJSWAP stats occasionally
 - Set OBJSWAP=X'02' and OBJSTMIN=100 (or bigger) in production



**Small and Efficient.
Still the 204 way with O-O.**

Any questions?



Sirius Software, Inc.