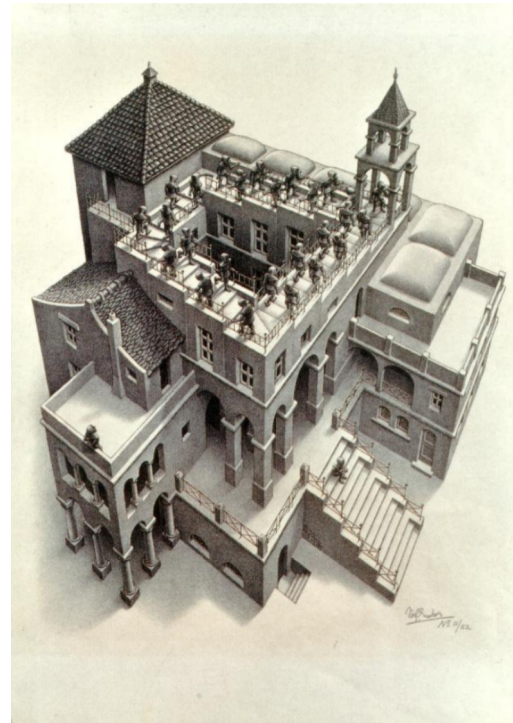


Recursion and Model 204 and Recursion and Model 204 and Recursion and Model 204 and ...

Alex Kodat
Sirius Software Inc.



Sirius Software, Inc.

Performing Repetitive Tasks in a Program

- This is what computers excel at
- Two basic approaches
 - Iteration
 - Recursion



Iteration

- Requires some kind of loop structure:

```
for %i from 1 to %numberOfTimes  
    ... repetitively executed code  
end for
```

- Many kinds of loops
 - For Each Record
 - For Each Value
 - Repeat Forever
 - Loops over members in other languages
- The most common kind of repetition in most software



Recursion

- A function, subroutine, method, that invokes itself
 - Might happen indirectly, i.e. A invokes B which invokes A
- Requires some kind method structure:

```
subroutine foobar(%mumble)
  ... repetitively executed code
  call foobar(%whatever)
  ... more repetitively executed code
end for
```

- Recursion can also apply to other things
 - An XML document can contain a document with the same structure, for example
 - Think Escher



Iteration vs. Recursion

- In general, any problem solvable by one, solvable by the other
 - But each better at certain problems
- Iteration generally less expensive
 - We'll talk about why, in a bit
- Iteration more common in most software
- There are, however, languages that don't support iteration and only support recursion
 - LISP is a prime example
 - Afficionado's swear this produces more reliable/maintainable code
 - True confession: I have no clue if they're right – they might be



A Factorial Function Using Iteration and Recursion

Recursion

```
function factorial(%number is float) -  
    is float  
  
if %number eq 1 then  
    return 1  
else  
    return %number *  
        %factorial(%number - 1)  
end if  
  
end function
```

Iteration

```
function factorial(%number is float) -  
    is float  
  
%i        is float  
%product  is float  
  
- %product = 1  
  for %i from 1 to %number  
    %product = %product * %i  
  end for  
  
  return %product  
  
end function
```



Iteration vs. Recursion: The Score so Far (cont..)

- Recursive code smaller. Advantage recursion.
- No local variables need to be defined. Advantage recursion.
 - This is probably a large aspect of the claim that recursive coding is more reliable a maintainable since the fewer variables used, the fewer variables that can get clobbered
- Recursive code almost certainly uses more storage
 - To hold the return points for calls and the intermediate %number's. Advantage iteration.



Iteration vs. Recursion: The Score so Far (..cont)

- Recursive code will almost certainly be more expensive. Advantage iteration.
 - Hard to do a call as cheaply as an increment
 - On 204 the recursive factorial was 88% slower for 20!
 - I was surprised: I thought it would be worse
- Recursive code requires creating a function while iteration could be done inline
 - While perhaps for something generic like Factorial, you wouldn't (do inline), many one-off algorithms you might not
 - So no need to come up with unique/meaningful names for methods
 - Ephemeral methods blunt this disadvantage
 - Still... Advantage Iteration



When is Recursion Unavoidable?

- When there is a one to many relationship among like data
 - For example, while processing parents you have to process children, and then, maybe, their children
 - A tree structure
 - Since there is no simple way to linearly process the data
- Usually you'll know when you need it
 - You don't need to go looking for excuses to use recursion
- If your recursive calls are followed by no, or very little code, recursion is probably unnecessary
 - Though maybe still makes the code tidier
- If your recursive methods/routines don't have many local variables, recursion is probably unnecessary



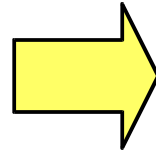
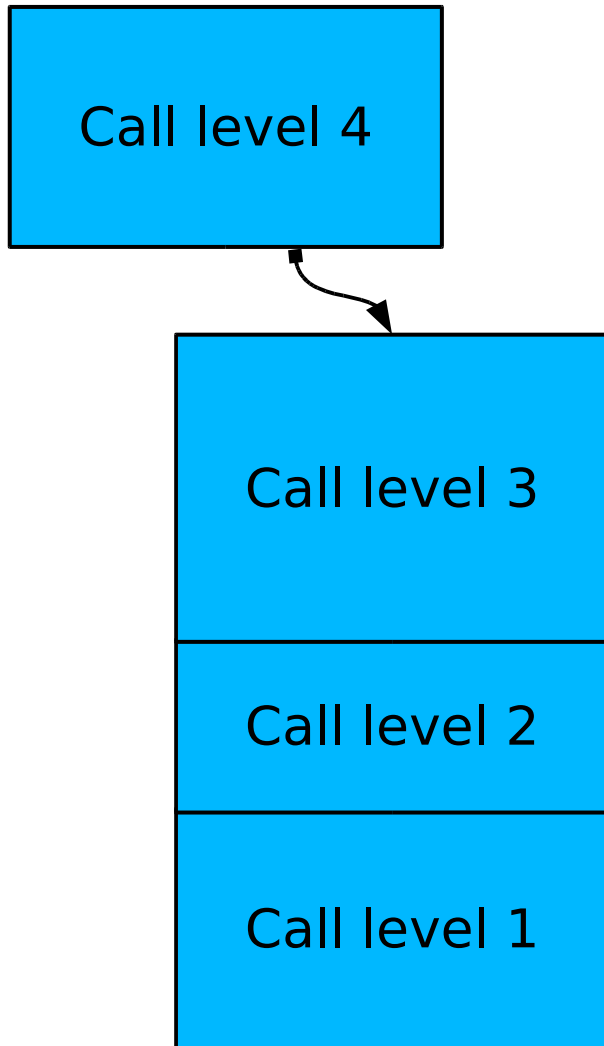
So How Does Recursion Work?

- At every level of recursion, certain information needs to be saved
 - At the very least, the return point
 - ➔ Since a recursive method, like any other method, can be called from lots of different places and each iteration must return to the correct caller
 - Input parameters
 - ➔ Since it would be nasty if a recursive call clobbered our input parameters
 - Local variables
 - ➔ Since it would be nasty if a recursive call clobbered our local variables
- Area to save this stuff is generically called a “stack”

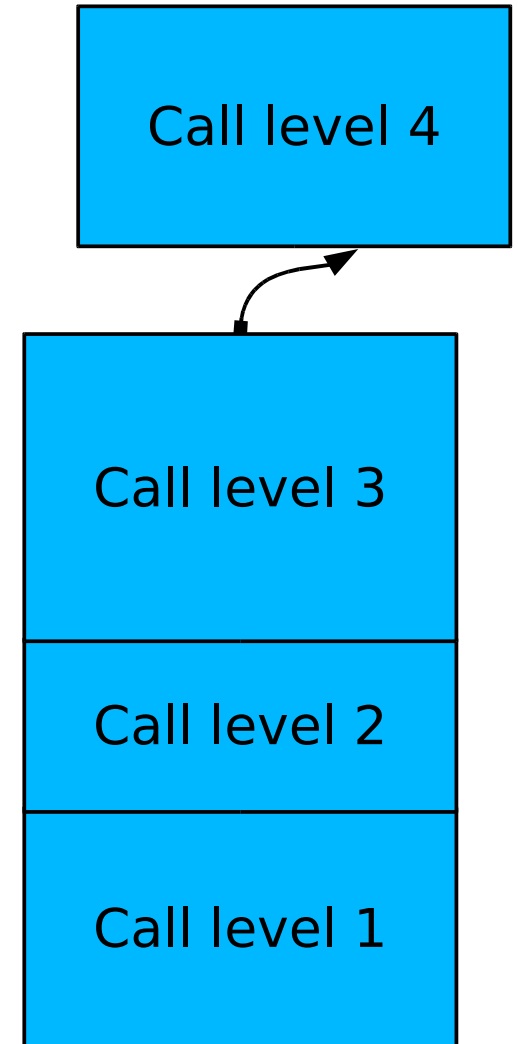
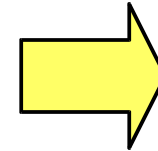
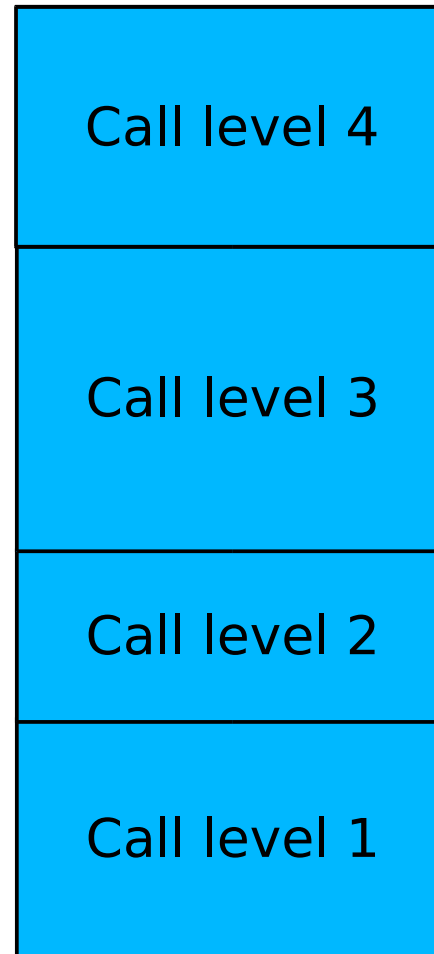


A Stack


A Call



A Return



The Stack in Model 204

- No single stack in Model 204
 - Internal call stack is called Push Down List, or PDL
 - Used by explicit and implicit UL calls
 - UL calls also stack things in VTBL and STBL
 - Complex subroutine call string expression results stacked in STBL
 - Methods stack local variable strings in STBL
 - So three stacks for the price of one
- Sadly, none of the stack high-water-marks are well maintained
 - Note to Sirius and CCA – someone should fix this one of these days
 - Sirius does have PDL stat fixup option
 - So take evaluation VTBL, STBL, PDL stats with a grain of salt 



Recursion with Simple Subroutines

- Simple subroutines have no parameters or local variables of their own
- So only thing to stack is their return point
 - Which is done correctly (duh)
- So not very useful for recursion
 - Useful for moving a complex block of code out of mainline code to make mainline code easier to read
 - Useful for sharing of code that has heavy local variable access but is used in more than one place



Recursion with Complex Subroutines

- Return point correctly stacked (duh)
- Parameters sort of sometimes stacked
- Local variables not stacked
- So some recursion possible, but can be very confusing



Complex Subroutine That Stacks Correctly

b

```
subroutine fibonacci(%previous is float, %current is float)
```

```
  if %current lt 100000 then
```

```
    call fibonacci(%current, %previous + %current)
```

```
  end if
```

```
  print %current
```

```
end subroutine
```

```
call fibonacci(0, 1)
```

```
end
```



Complex Subroutine That Doesn't Stack Correctly

b

```
subroutine fibonacci(%previous is float, %current is float)
```

```
%next is float
```

```
%next = %previous + %current  
if %next lt 100000 then  
    call fibonacci(%current, %next)  
end if
```

This is the problem
Local variables
aren't stacked.

```
print %next
```

```
end subroutine
```

```
call fibonacci(0, 1)
```

```
end
```



Local Variables Completely Static in Complex Subroutines

- So makes complex subroutines of limited utility for recursion
 - Since one rule of thumb for recursion unnecessary is not many local variables
- This is even a problem in non-recursive cases
 - Since values of variables left over from previous iterations
 - Classic source of bugs



Complex Subroutine Bug: Forgot to Initialize Counter at Each Call

```
subroutine totalPayments(%recno is float, %result is float output)
```

```
%total is float initial(0)
```

```
in file foo for record number %recno
```

```
o: for each occurrence of payment
```

```
    %total = %total + value in o
```

```
end for
```

```
end for
```

```
%result = %total
```

```
end subroutine
```

Note: The Initial clause operates at compile-time so does not help



Sirius Software, Inc.

SOAP ULI Methods – Designed with Recursion in Mind (cont..)

- All parameters and variables stacked when recursion detected
 - So no stacking on first level call
 - Since local string variables have an STBL component, this means that significantly more stacking is now done in STBL
 - So method recursion more table-intensive than simple or complex subroutine recursion
 - And possibly more CPU-intensive
 - ➔ But highly efficient machine-language avoids UL hoop-jumping to deal with lack of stacking



SOAP ULI Methods – Designed with Recursion in Mind (..cont..)

- Functions and properties allowed in addition to subroutines
 - Eliminating (most of the) need for output parameters which complicate recursion
 - Can be included in the middle of an expression, simplifying recursive calls:

```
return %input * %factorial(%input - 1)
```

- In some cases SOAP ULI has to stack intermediate expression results



SOAP ULI Methods – Designed with Recursion in Mind (..cont..)

- All local variables set to initial values at method entry
 - Trading off fast machine-language variable initialization for fewer (because not all variables need to be initialized) but more expensive UL initialization assignments
 - Preventing bugs that only happen on later invocations of a method
 - Preventing poor programming practice of hiding persistent data inside methods
 - Persistent data should be held in a class Public [Shared] or Private [Shared] block
 - Static variable allowed and not stacked
 - Local classes (coming soon) will provide another persistence option



SOAP ULI Methods – Designed with Recursion in Mind (..cont)

- All local object variables set to null at method exit
 - Meaning locally created objects immediately cleaned up on exit
 - Meaning references to objects in methods that have returned don't prevent objects from being discarded automatically (because of a zero reference count)
- All local longstring CCATEMP data cleaned up on exit
 - Longstrings are kind of objects, anyway



What if an Algorithm Might Recurse Hundreds or Thousands of Times?

- Make your servers much bigger
 - Let's say you stack 500 bytes of data on each iteration (that's a lot) and might recurse 5,000 times. That requires a mere extra 2.5M server size
 - Maybe that was scary 10 years ago, but is it still?
- Roll your own recursion using objects



Rolling Your Own Recursion Using Objects

- Define a class that contains all the data specific to the recursion levels
- When you get to a recursion point, push a reference to the current object onto an ArrayList
- After pushing, create a brand new object or a copy of the old object as the new current object
 - Depending on whether more data is “inherited” from previous recursion level or more is new
- Push and Pop Arraylist methods available in Sirius Mods 7.0
 - Though not hard to roll your own
 - Queue and Dequeue also available



Rolling Your Own Recursion With Objects

Example:

```
class telephoneNumber
  public
    variable numberCalled    is arrayList of object phone call
    variable checked         is boolean
    variable beingChecked   is boolean
    variable suspiciousness is float
    ...
  end public
end class

...
%currentNumber is object telephoneNumber
%numberStack  is arrayList of object telephoneNumber

...
* Need to check number called
%numberStack:push(%currentNumber)
%currentNumber = new
... and do process for new number
%currentNumber = %numberStack:pop
if %currentNumber ne null then
  ... pick up where we left off
end if
```



Rolling Your Own Recursion

- Stacking is done in CCATEMP
 - ArrayList of object references mostly in CCATEMP
 - When number of stacked objects exceeds object slots in VTBL/STBL objects are swapped to CCATEMP
- So ideal for intensively recursive application
- But the code will likely be somewhat more difficult to understand
- Maybe some day, method recursion will (optionally) stack data in CCATEMP
 - Let us know if this is important to you



Summary

- It's worth understanding recursion
 - And now you do
- Janus SOAP ULI provides some new capabilities for recursive programming



Questions or comments?

