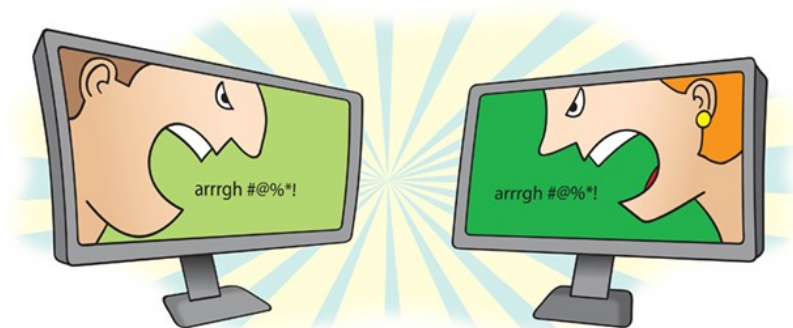


All Programming is Local



Alex Kodat
Sirius Software Inc.

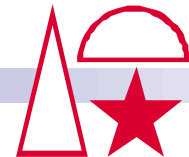
Simple Subroutines

- The original way to encapsulate common code in User Language
- No parameters specifiable
- No result value possible
- Can only be inside outermost (Begin/End) level
 - Can't be inside of methods or subroutines
- Shares scope (variables) with outermost (Begin/End) level
 - Can be useful
 - But can be a pain as defeats some of the purpose of encapsulation



Complex Subroutines

- The new way to encapsulate common code in User Language... well, new 20 years ago
- Parameters specifiable
 - But no optional or named parameters
- No result value possible
- Can only be inside outermost (Begin/End) level
 - Can't be inside of methods or subroutines
- Doesn't share scope (variables) with anyone else
 - Though Common variables shared, of course



Class Methods

- The new way to encapsulate common code in User Language... well, new 6 years ago
- Parameters specifiable
 - Including optional and named parameters
- Result value possible (functions)
- Can only be inside a class
 - Can't be inside of methods or subroutines
- Doesn't share scope (variables) with anyone else
 - Though Common variables shared, of course



Introducing Local Methods

- The newest way to encapsulate common code in User Language
- Parameters specifiable
 - Including optional and named parameters
- Result value possible (functions)
- Can be inside outermost (Begin/End) level or inside other methods
 - Including other local methods
- Can share scope (variables) with container or not
 - Up to method coder



Defining a Local Method

- Starts with Local keyword
- Followed by Subroutine, Function, or Property
- Then followed by method name
 - (<className>):<methodName> syntax for local enhancement methods
- Then followed by parameters, result datatype and other method qualifiers
 - Most, but not all qualifiers available on class methods available here, along with some new ones
 - We'll talk about this later
 - Generally no need to Declare method before defining it



Example of Very Boring Local Subroutine

b

```
local subroutine foo
  printText Here I am
end subroutine
```

```
call %foo
%(local):foo
```

```
end
```



Local Subroutine

- Called either via `Call %<subroutineName>` or `%(local):subroutineName` or even `%(local):subroutineName`
- Has its own scope (variables)
- Name cannot match name of %variable in containing scope



Example of Slightly More Interesting Local Subroutine

b

```
%foo is float
```

```
local subroutine showList(%sl is arraylist of longstring)
  %i    is float
  for %i from 1 to %sl:count
    printText {%i:right(5)}: '{%sl(%i)}'
  end for
end subroutine
```

```
%al    is arraylist of longstring
```

```
'''
%(local):showlist(%al)
```

```
end
```



More Local Subroutine Tidbits

- Local subroutines can have parameters
 - Optional and named parameters supported, of course
- Local subroutines can have method objects
 - Indicated in method name as `(<classname>):<methodname>`
 - So previous example not quite as nice as it could be
- Enhancement method syntax not required to invoke subroutine
 - Because it's local so no confusion about where it's coming from



Previous Example Done Better

```
b
%foo is float

local subroutine (arraylist of longstring):showList
  %i is float
  for %i from 1 to %this:count
    printText {%i:right(5)}: '{%this(%i)}'
  end for
end subroutine

%al is arraylist of longstring
...

%al:showlist

end
```



Local Functions and Properties

- Functions return a value
 - Can also be declared Callable, like class functions
 - Intrinsic local functions quite common
- Properties can return or be set to a value



Example of a Local Intrinsic Function

b

```
local function (string):numberOfVowels is float
    %i      is float
    %vowels is float
    %this = %this:toUpper
    for %i from 1 to %this:length
        if %this:char(%i):positionIn('AEIOU') then
            %vowels = %vowels + 1
        end if
    end for
    return %vowels
end function
```

```
printText {~} = {'The lunatic is on the grass':numberOfVowels}
```

end

* Prints

```
'The lunatic is on the grass':numberOfVowels = 8
```



Method Hiding

- A local enhancement method on a class will “hide” any eponymous method in that class
 - This prevents new methods added to a class from breaking local enhancement methods
- The hidden method can be unhidden using the Local Alias statement



Example of a Hidden System Method

This is NOT Recommended

b

```
local subroutine (stringlist):add(%what is longstring)
  local alias (stringlist):addItem for add
  %this:addItem('' with %what with '')
end subroutine
```

```
%sl is object stringlist
```

```
%sl = new
```

```
%sl:add('Hello')
%sl:add('Goodbye')
%sl:print
```

```
end
```



Local Aliases for Class Enhancement Methods

- Allow access to enhancement method without having to specify (+<classname>) in front of the method name
 - Useful if a **lot** of references to an enhancement method in a scope
 - Makes code prettier
- Can be used to access the enhancement method by a different name
 - Not recommended – likely to cause confusion



Example of a Local Alias for a Class Enhancement Method

```
class util
  public shared
    function (stringlist):everyOtherItem is object stringlist
  end public shared
  function (stringlist):everyOtherItem is object stringlist
    %i      is float
    %outsl  is object stringlist
    %outsl = new
    for %i from 1 to %this:count by 2
      %outsl:add(%this(%i))
    end for
    return %outsl
  end function
end class
```

```
local alias (stringlist):everyOtherItem for (+util)everyOtherItem
local alias (stringlist):eo              for (+util)everyOtherItem
```

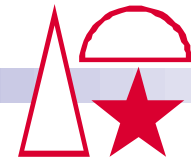
```
  ...
%sl  is object stringlist
%sl2 is object stringlist
```

```
  ...
%sl2 = %sl:(+util)everyOtherItem
%sl2 = %sl:everyOtherItem
%sl2 = %sl:eo
```



Nested Local Methods

- Local methods can be contained inside of class methods or other local methods or even complex subroutines
- So can only be called inside the containing scope
 - And sometimes by other methods inside the same scope
 - And always recursively (method can always call itself)
 - So, be careful



Example of Nested Local Method

```
local function (stringlist):everyOther is object stringlist
```

```
    local function (string):everyOther is longstring
```

```
        %i is float
```

```
        %out is longstring
```

```
        for %i from 1 to %this:length by 2
```

```
            %out = %out with %this:char(%i)
```

```
        end for
```

```
        return %out
```

```
    end function
```

```
%i is float
```

```
%outsl is object stringlist
```

```
%outsl = new
```

```
for %i from 1 to %this:count by 2
```

```
    %outsl:add(%this(%i):everyOther)
```

```
end for
```

```
return %outsl
```

```
end function
```



Exposed Local Methods

- Indicated by keyword `Expose` on method declaration
- Indicates that variables and methods in containing scope can be accessed in local method
- However, a local variable inside an exposed method can hide containing scope method
- Provide an alternative to simple subroutines



Example of an Exposed Local Method

```
b

%a is float
%b is float

local function (float):add is float expose
    return %this + %a + %b
end function

%a = 2
%b = 3

%a = %a:add
%b = %b:add

printText {~} = {%a}, {~} = {%b}

end
```



Advantages of Exposed Local Methods Over Simple Subroutines

- Locally scoped variables not seen in containing scope
- Allows method object and parameters
 - Including named and optional parameters
 - So natural O-O syntax on calls
- Can return value (functions/properties)
- Local variables auto-initialized on every call
- Local variables always stacked on recursion
- Can be nested inside other methods/complex subroutines
- Never write a simple subroutine again



Common Subroutines

- Intended to allow backward-compatible conversion of complex subroutine to method
- Keyword Common used instead of Local
- Must be in level 0 scope like complex subroutine
- Called like complex subroutine:
 - Call <subroutinename>



Example of an Common Subroutine

```
b  
  
common subroutine foo(%x is float, %y is float)  
    printText {~} = {%x * %y}  
end subroutine  
  
call foo(13, 77)  
  
end
```



Advantages of Common Subroutines Over Complex Subroutines

- Allows optional and named parameters
- Local variables auto-initialized on every call
- Local variables always stacked on recursion
- So very useful if need to add a parameter to heavily used complex subroutine
 - Simply change Subroutine statement to Common Subroutine
 - Add optional (maybe named) parameter
 - And you're off to the races
 - No need to change existing callers



Common Properties

- Intended to allow a Common variable to be turned into a property
- Keyword Common used instead of Local
- Must be in level 0 scope like complex subroutine
- Used like a variable
 - Variable must be declared Common
 - Can't use as Output variable



Example of an Common Property

b

```
common property abc is float
  %abcShadow is float shared
  get
    return %abcShadow
  end get
  set
    assert %abc gt 0
    %abcShadow = %abc
  end set
end property
```

```
%abc is float common
```

```
%abc = 22
print %abc
%abc = -11
print %abc
```

end



Reasons To Use Common Properties

- Enforce rules for setting values
- Create common variables with derived values
 - Cleaner access to globals?



Conclusions

- Local methods a very useful general purpose facility
 - You don't even need to use “O-O” to take advantage of them
 - Intrinsic and XmlDoc local methods especially useful
- Common subroutines are a nice facility for migrating complex subroutines to take advantage of new Janus SOAP ULI facilities



Questions? Comments?



Sirius Software, Inc.