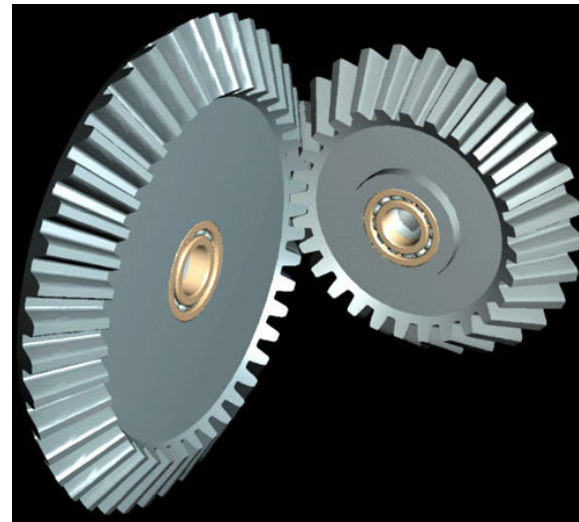


Understanding Methods

Alex Kodat
Sirius Software Inc.



What Are Methods?

- Code that performs a particular function
- In some environments **all** code is in one method or another
- In User Language/Janus SOAP come code is not inside a method
 - Outer-most request level
 - Complex and simple subroutines
- Always in a class



Object Methods

- Operate on a single instance of a class
 - Typically, retrieve and update values of class variables for an object
 - The object to which the method is being “applied” is an implied input parameter
 - Sometimes accessed via %this
 - Sometimes access implicitly – more on this later
- Must be *declared* inside of class public/private block
- If ever used, must be *defined* (code specified) inside of class block
 - Definition includes header which (more or less) repeats declaration



The Three Types of Methods (in Janus SOAP)

- Subroutines
 - Return no output, cannot be “set”
- Functions
 - Return an output, cannot be “set”
- Properties
 - Return an output, can be set



Declaring Methods

```
class customer
  public
    variable    crn          is string len 10
    variable    surname      is string len 64
    subroutine  sendletter(%contents is object stringList)
    function    monthlyPayment is float
    property    income        is float
  end public
end class
```



Declaring a Method

```
class customer
  public
    variable    crn        is string len 9
    subroutine  setCrn(%inCrn is string len 9)
  end public

  subroutine  setCrn(%inCrn is string len 9)

    assert $len(%inCrn) = 9
    %this:crn = %inCrn

  end subroutine
end class
```



Invoking a Method

- Invoked by object variable, followed by colon, followed by method name
- If a method has an output object, that object can also be followed by a colon and method
- Parameters to methods enclosed in parentheses just like complex subroutine parameters



Invoking a Method

```
class customer
  public
    variable    crn        is string len 9
    subroutine  setCrn(%inCrn is string len 9)
  end public

  subroutine  setCrn(%inCrn is string len 9)

    assert $len(%inCrn) = 9
    %this:crn = %inCrn

  end subroutine
end class

%cust    is object  customer

%cust = new

%cust:setCrn('987654321')
```



Functions

- Return an output
 - Simple (Float, Fixed, String, Longstring) or Object
 - Return value/object specified on return statement
- Look exactly like readOnly properties (more on these later)
 - Function implies more processing
 - And possibly side-effects (modification of the input object)
 - Function usually has parameters (but doesn't have to)
 - But don't worry too much about the difference



Example of a Function

```
class customer
  public
    variable      crn          is string len 9
    function      countCharacter(%char is string len 1) is float
  end public

function      countCharacter(%char is string len 1) is float

  %i          is float
  %count      is float

  for %i from 1 to $len(%this:crn)
    if $substr(%this:crn, %i, 1) eq %char then
      %count = %count + 1
    end if
  end for

  return %count

end function

end class
```



Invoking the Function

```
class customer
  public
    variable      crn          is string len 9
    function      countCharacter(%char is string len 1) is float
  end public

  function      countCharacter(%char is string len 1) is float
    ...
  end function
end class

%cust          is object customer
%onesAndTwos   is float
  ...

print %cust:countCharacter('1')

%onesAndTwos = %cust:countCharacter('1') + %cust:countCharacter('2')
```



Properties

- Look very much like variables
 - In fact they're isomorphic to variables
 - Can change a variable to a property and vice versa, without changing calling code
 - Except for properties with parameters
 - These look more like collections (more on these later)
- Can return values like a function
 - Except a WriteOnly property
- Can be set to a value
 - Except a ReadOnly property



Property Methods

- Each property actually two methods
 - A Get method and a Set method
- Get method must return value via Return statement
 - Just like a function
 - Doesn't exist for WriteOnly methods
- Set method input (set value) is variable with same name as property



Example of a Property

```
class customer
  public
    variable      crn          is string len 9
    variable      birthdate    is fixed
    property      ageInDays    is fixed
  end public

  property ageInDays is float

    get
      return $sir_datend - $sir_date2nd(%birthdate, 'YYYYMMDD')
    end get

    set
      %birthdate = $sir_nd2date($sir_datend - %ageInDays, 'YYYYMMDD')
    end set

  end property
end class
```



Invoking the Property

```
class customer
  public
    variable      crn          is string len 9
    variable      birthdate    is fixed
    property      ageInDays    is fixed
  end public
  ...
end class

%cust  is object customer
%cust  = new

%cust:birthDate=('19590813')
print %cust:ageInDays

%cust:ageInDays = 1500
print %cust:birthdate
```



ReadOnly and WriteOnly Properties

- While properties are analagous to variables, some times you want the equivalent of readOnly variables
 - While behavior identical to a function, philosophically different
- WriteOnly unusual, but provides the equivalent of variables that you can only set



An Example of a ReadOnly Property

```
class customer
  public
    variable      crn          is string len 9
    variable      birthdate    is fixed
    property      ageInDays    is fixed  readOnly
  end public
  property ageInDays is float
  get
    return $sir_datend - $sir_date2nd(%birthdate, 'YYYYMMDD')
  end get
end property
end class

%cust  is object customer
%cust  = new

%cust:birthDate=('19590813')
print %cust:ageInDays
```



Private Variables

- Contained in a Private block, like public variables contained in a Public block
 - Public block has exact same format as private block
- Private variables only accessible from methods inside the class
 - So useful as a mechanism for hiding internal information from users of the class
 - Properties are often shadowed by private variables
 - Extremists say that there should be **no** public variables
 - All variables should be accessed via get/set methods
 - Properties makes this unnecessary



Using Private Variables In a Property

```
class customer
    public
        property      birthDate      is string len 8
    end public
    private
        variable      prvBirthdate is float
    end private

    property      birthDate      is string len 8
        get
            return $sir_nd2date(%this:prvbirthDate, 'MMDDYYYY')
        end get
        set
            %newBirthDate is float
            %newBirthDate = $sir_date2nd(%birthDate, 'YYYYMMDD')
            assert %newBirthDate gt $sir_date2nd('18800101', 'YYYYMMDD')
            %this:prvbirthDate = %newBirthDate
        end set
    end property

end class
```



More About Methods

- A null method object is almost always a request canceling error
 - Oddball *AllowNullObject* keyword on method declaration overrides this
- A null input object parameter to a method is usually a request canceling error
 - *AllowNull* keyword on parameter declaration overrides this
- *%this:* does not need to be specified
 - It can be replaced with % and the *this:* is implied
 - Is this a good or bad thing? You decide.
- Output parameters supported just like complex subroutines
 - Though more avoidable



Example of a Implicit *this*:

```
class customer
  public
    variable      crn          is string len 9
    function      countCharacter(%char is string len 1) is float
  end public

function      countCharacter(%char is string len 1) is float

  %i          is float
  %count      is float

  for %i from 1 to $len(%crn)
    if $substr(%crn, %i, 1) eq %char then
      %count = %count + 1
    end if
  end for

  return %count

end function

end class
```



Constructor Methods

- Used to do work when object created (instantiated)
- Since objects typically created with New, a New constructor is the typical constructor
- Constructors can have parameters
- Input to constructor is newly created object



Example of a Constructor

```
class customer
  public
    variable      crn          is string len 9
    variable      creationTime is float
    constructor new
  end public

  constructor new

    %creationTime = $sir_datens

  end constructor
end class
```



Example of a Constructor with Parameters

```
class customer
  public
    variable      crn          is string len 9
    variable      creationTime is float
    constructor new(%crn is string len 9)
  end public

  constructor new(%crn is string len 9)

    %creationTime = $sir_datens
    %this:crn = %crn

  end constructor
end class

%cust      is object customer

%cust = new('123456789')
```



Shared Methods

- Do not operate on a specific instance of a class
 - So no %this defined inside
- Usually perform some basic housekeeping functions for a class
- Declared inside Public Shared or Private Shared block



Example of a Shared Method

```
class customer
  public
    variable      crn          is string len 9
    variable      creationTime is float
    constructor new(%crn is string len 9)
  end public
  private shared
    variable      prvFirstCustomer is object customer
  end private shared
  public shared
    function firstCustomer is object customer
  end public shared
  function firstCustomer is object customer
    return %(customer):prvFirstCustomer
  end function
end class

%cust      is object customer
...
print %(customer):firstCustomer:crn
```



Factory Methods

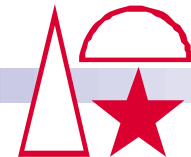
- Methods that create a new instance of an object
 - Or, occasionally return an existing object
- Usually shared methods
- Can look like constructors to the calling code
 - But different from constructors in that they're call **before** new object instantiated



Example of a Factory Method

```
class customer
  public
    variable      crn          is string len 9
    variable      creationTime is float
    constructor new(%crn is string len 9)
  end public
  public shared
    function newCustomer is object customer
  end public shared
  function newCustomer is object customer
    %cust      is object customer
    %cust = new
    %cust:crn = '???????????'
    return %cust
  end function
end class

%cust      is object customer
...
%cust = newCustomer
```



Optional/Default Parameters

- Sometimes a large percentage of invocations of a method use the same value for a parameter
 - Often 0 for fixed or float variables
- Parameter declaration can indicate *Optional* to use standard default value
 - 0 for Float and Fixed, null string for String and Longstring, null for objects
- Fixed, Float, String, and Longstring parameter declaration can indicate an alternate default value
- Presence of parameter can be tested with *Is Present/Is Not Present* clause



Example of Optional/Default Parameters

```
class customer
    ...

    function padCrn(%len is fixed optional,          -
                  %char is string len 1 default '.' -
                  is longstring)

        if %len is not present then
            %len = $len(%crn) * 2
        end if

        return $lstr_left(%crn, %len, %char)

    end function

end class

%cust      is object customer

%cust = new('123456789')

print %cust:padCrn(, '*')
```



AllowNull Parameters

- Usually, a method with an object parameter expects the passed object to be there (non-null)
- With Janus SOAP ULI, passing a null object as a parameter is a request cancelling run-time error at the time of the call
- But sometimes a method can handle a null object parameter
 - In such a case specify *AllowNull* on the parameter declaration
 - All Optional object parameters are assumed to be AllowNull
 - Maybe a required AllowNull parameter is a strange beast?



Example of AllowNull Parameter

```
class customer
  ...

  function isRelatedTo(%otherCustomer is object customer allowNull) -
    is boolean

    if %otherCustomer is null then
      return 0
    end if
    ...          check for customer relationship
  end function
end class

%cust      is object customer
%someone   is object customer
...
if %cust:isRelatedTo(%someone) then
  ...
```



The Callable Keyword

- Some functions are really subroutines in disguise
 - They perform work like a subroutine
 - But return an informational value that might not be interesting to the caller of the function
- In such a case, use the *Callable* keyword
 - Allows the function to be called without a result variable



Example of a Callable Function

```
class customer
    ...
    function addActivity(%what is object activity) is float callable
        ...
        return %numberOfPendingActivities
    end function
end class

%cust is object customer
...
%cust:addActivity(%addChild)
...
if %cust:addActivity(%addChild) gt %maxPendingActivities
    %cust:completePendingActivities
end if
```



Named Parameters

- Allows parameters to be specified by name, instead of position, on calls
 - Both for system classes and for User Language classes
- Not available for all parameters
 - At discretion of method coder
- Along with optional parameters, the SOAP ULI answer to overloading
 - But superior to overloading



An Example of Named Parameters

```
class customer
    public
        ...
        function getId(%surname      is string len 32 namerequired, -
                        %givenName   is string len 32 optional,      -
                        %state       is string len  3 optional,      -
                        %city        is string len 32 optional)

        ...
    end public
end class

...
%cust      is object customer
...
%id = %cust:getId(givenName='Sheila', surname=%ln, state='WA')
```



Some Features of Named Parameters ...

- Percent symbol appears in declaration of method and internal references, but does **not** appear in parameter name in method invocation
- Parameter name can be required on invocations (*nameRequired*) or just allowed (*nameAllowed*)
 - All parameters after first *nameAllowed* parameter are also (implicitly) *nameAllowed*
 - All parameters after first *nameRequired* parameter are also (implicitly) *nameRequired*
 - Think about it
- Named parameters can be specified in any order on invocation



Some Features of Named Parameters (cont.) ...

- Named parameters can be preceded by one or more non-named parameters
 - Of course, non-named parameters have names on the declarations, the names just can't be specified on an invocation
- Named parameters will generally be optional, but don't have to be
 - *Optional* or *Default* must be specified for optional named parameters
- Indicated by equals sign after first token in method argument
 - A syntactic “anomaly”



Some Features of Named Parameters (cont.)

- Available on all kinds of methods: subroutines, function, and properties
 - But not on so-called complex subroutines (non-class-related subroutines)



Some Sample Named Parameter Invocations

```
%id = %cust:getId(givenName='Sheila', surname=%ln, state='WA')
```

is the same as

```
%id = %cust:getId(surname=%ln, state='WA', givenName='Sheila')
```

is the same as

```
%id = %cust:getId(surname=%ln, givenName='Sheila', state='WA')
```

...

```
%id = %cust:getId('Sheila', %ln, state='WA')
```

...

```
%id = %cust:getId(, %ln, state='WA')
```



When Should One Use Named Parameters?

- When no obvious order to parameters
 - Especially useful for people reading code
- When lots of optional parameters
 - Avoids comma hell
- When arguments are often (cryptic) literals
 - Booleans are the most common example

```
%cust:addProvider(true, false)
```

is not as easy to understand as

```
%cust:addProvider(parent=true, isCustomer=false)
```



When Shouldn't One Use Named Parameters?

- When parameter semantics painfully obvious from method name and/or context

```
%cust:changeState(%state)
```

is good, but the following silly (and jarring to read)

```
%cust:changeState(state=%state)
```



Some Tips on Named Parameters

- Choose good parameter names (duh)
 - Do this even for non-named parameters as the names appear in the templates and are often used as mnemonics by programmers
- When in doubt about parameter position make it *NameRequired*.
 - You can always make it *NameAllowed* later
- If not sure whether you want to allow names for a parameter, don't
 - You can always make it *NameAllowed* later
- Use them generously



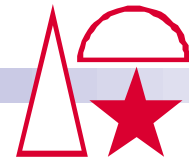
A Problem With Parameter Names

- Inside methods you want parameters to be obvious by their names
 - So you adopt a standard that all parameters start with, say, *parm*.

```
if %parm.state = 'ACT' and %parm.surname = 'LITTLETON' then
    %weeklyPayment = %weeklyPayment + 3 * %parm.increase
else
    %weeklyPayment = %weeklyPayment + %parm.increase
end if
```

- But this produces ugly method declarations and invocations

```
subroutine adjustWeekly(%parm.increase nameAllowed, %parm.state, -
    %parm.surname) 'LITTLETON' then
...
%cust:adjustWeekly(parm.increase=%increase, parm.state='ACT', -
    parm.surname=%surname)
```



A Solution to the Problem - The InternalNames Block

- Appears at the start of a method definition
 - Not the declaration
 - Before Get or Set methods for a property
- Maps names used internally in method to parameter names
 - Can even re-map *%this*
- Percent required in both internal and external names
- Useful even on non-named parameters because it makes the method declarations easier to read



InternalNames Block Example

```
subroutine adjustWeekly(%increase nameAllowed, %state,
                        %surname)

    internalNames
        %parm.increase      is %increase
        %parm.state         is %state
        %parm.surname       is %surname
    end internalNames

    ...
    if %parm.state = 'ACT' and %parm.surname = 'LITTLETON' then
        %weeklyPayment = %weeklyPayment + 3 * %parm.increase
    else
        %weeklyPayment = %weeklyPayment + %parm.increase
    end if

    ...
end subroutine

...
%cust:adjustWeekly(increase=%increase, state='ACT', surname=%surname)
```



System Classes/Objects

- Embedded in Model 204 nucleus
 - Like \$functions
 - Written in Assembler
 - So very efficient
- Class declarations not in User Language
 - Need to read the documentation to find out what's available
- Contain no public variables
- But lots of methods/properties



System Object Example

```
%list      is object stringList
```

```
%list = new
```

```
%list:add('Canberra')
```

```
%list:add('Cambridge')
```

```
%list:add('Marblehead')
```

```
%list:add('Oxford')
```

```
%list:sort('1,50,A')
```

```
%list:print
```



StringList Objects

- Use instead of \$lists
- Compile-time data type checking
- Better instantiation model
 - Not tied to \$listnew statement
- Nicer method names
 - Future functionality enhancements are likely to be for StringLists and not \$lists
- MoveTold and MoveFromId methods aid in migration



Local Use of \$list as StringList

* We received %list as a \$list

%sl is object stringList

%sl = new

%sl:moveFromid(%list)

%sl:add('Canberra')

%sl:add('Cambridge')

%sl:add('Marblehead')

%sl:add('Oxford')

%sl:sort('1,50,A')

%sl:print

%sl:moveToId(%list)



That's enough about methods

Any questions?

