upstate.edu

Lesson 2: #define and #evaluate

35-45 minutes

Differences between XSLT and Velocity

- With respect to the differences between XSLT and Velocity, Hannon Hill maintains that "essentially any script that can be written in one language can be written in the other" (see Is Velocity "better" than XSLT (or vice-versa)?

)

- This is true ONLY if we consider the transformation of block data
- Once we move beyond block data transformation, we will learn that Velocity is much more powerful than XSLT
- There are many things that can be done with Velocity but not with XSLT:
- In XSLT, we can only create constants; no variables can be created, not to say the creation of global variables
- There are only a few data types in XSLT, whereas Velocity can deal with any Java data types
- These two facts imply that even if we can use reflection to invoke Java methods to retrieve or create objects, these objects cannot be returned properly and stored in variables when using XSLT
- Therefore, using Java reflection is almost impossible in XSLT
- Besides, Velocity supports two important keywords: `define` and `evaluate`

- In this lesson, we will see how these two keywords enhance the power of Velocity tremendously
- In fact, both the intermediate course and advanced course show you what you can do only with Velocity

Code vs. String

- The Velocity engine will try to interpret a sequence of characters, or any parts of it, as Velocity code first
- This is particularly true when the sequence of characters appear between a pair of double quotes:

```
#set( $str = "#set(" )
```

When we try to submit code like this, we will see a `Failed to parse String literal at templateValidation` message; this makes sense only if we understand the fact that Velocity engine is parsing the line, not as a String, but as code. This is the process of code validation mentioned in Lesson 5: Macros.

- To further prove that a String is interpreted as code whenever possible, consider the following:

```
#set( $str = "#set($var=4)" )
$var            ##=> 4
$str.Class.Name ##=> java.lang.String
-$str -         ##=> - -
```

If the character sequence `"#set($var=4)"` is treated as a String, then the variable `$var` is not defined; on the other hand, if the character sequence is treated as code, then the variable is indeed defined and stores the value `4`. This code snippet does output `4`.

- In fact, both variables `$str` and `$var` are defined
- A String can also contain an invocation of a macro
- To see this work, I will use both `#define` and `#evaluate`:

```
#macro( myMacro )
    myMacro invoked
#end

#set( $str = "#define($callMyMacro)#myMacro#end" )
$callMyMacro

#set( $str = "#evaluate( ""#myMacro"" )" )
$str

#set( $str = "#myMacro" )
$str
```

This code snippet outputs `myMacro invoked` three times.

- Some notes on quotes:
- Within a pair of outmost double quotes, two adjacent double quotes can be used as an escaped sequence for the double quote

character:

```
#set( $str = """Hello, my name is John."" said
John." )
$str ##=> "Hello, my name is John." said John.

#set( $str = """" )
$str.length() ##=> 1
```

- The outmost pair of double quotes are part of the code, whereas the `""` sequence is part of the String literal
- Within a pair of outmost single quotes, two adjacent single quotes can be used as an escaped sequence for the single quote character:

```
#set( $str = 'His name is O''Connor.' )
$str ##=> His name is O'Connor.
```

- The use of quotes can get really interesting if we fully appreciate the difference between code and String
- We saw the following example in Lesson 10: Structured Data Nodes:

```
<img src="${map[ "slide-show-group;mul-
image;$foreach.index;selectimage" ][ $KEY_DATA ]}"
     alt="${map[ "slide-show-group;mul-
image;$foreach.index;altText" ][ $KEY_DATA ]}" />
```

  There are eight double quotes in this snippet; four of them are part of the code, and the other four are part of the String literal.

- A sequence of characters may not be meaningful when treated as code, especially when the code is evaluated to `null`; if that's the case, then the sequence will be output as a String
- Note that when such a String is assigned to another variable, the variable will store the String, not `null`:

```
#set( $str = "$var" )
$str.length() ##=> 4
```

  Here the variable `$str` stores a String of four characters. But if the variable `$var` is defined, then the result will be different:

```
#set( $var = 4 )
#set( $str = "$var" )
$str.length() ##=> 1
```

- In a `#set` statement, double quotes allow parsing and variables and directives will be parsed and validated (i.e., variables interpolated and directives executed), and single quotes force the String interpretation

Why `#define` and `#evaluate`?

- Both `#define` and `#evaluate` turn Strings into code
- Besides turning Strings into code, both directives can also execute the code at render time

- Combined with the techniques of String manipulation (String replacement, subStrings, etc.) and variable interpolation, we can generate code and execute it on the fly
- With this technique, variable names can be embedded with other variables, and a macro name can be computed dynamically and the corresponding macro invoked at render time
- We can write really succinct code when we can generate and execute code on the fly:

```
#set( $chanwTextSubTypes = [ "Radio", "PlainText",
"Checkbox", "Multiselect", "Wysiwyg", "Dropdown",
"Calendar", "Datetime" ] )
#foreach( $subType in $chanwTextSubTypes )
#set( $chanwCode = '#if( $chanwTextOptions.is' +
$subType + '())Subtype: ' + $subType + '#if(
$chanwWithBr )$BR#else$chanwNEWLINE#end#end' )
#evaluate( $chanwCode )
#end
```

  Without using `evaluate` in this snippet, we will need eight blocks of code of the following type:

```
#if( $chanwTextOptions.isRadio() )
Subtype: Radio
#if( $chanwWithBr )$BR#else$chanwNEWLINE#end
#end

#if( $chanwTextOptions.isPlainText() )
Subtype: PlainText
#if( $chanwWithBr )$BR#else$chanwNEWLINE#end
#end
...
```

- The use of code templates is made possible

The `#define` Directive

- The `#define` directive allows us to assign a block of code to a variable
- Whenever the variable is referenced, the block of code it is associated with will be executed
- The associated block of code can contain any valid code

A Simple Hello World Example

```
#define( $greetings )
Hello, World!##
#end
$greetings
$greetings
$greetings.Class.Name ##=>
```

`org.apache.velocity.runtime.directive.Block$Reference`

- The `#define` (and `#end`) is used to wrap up a String to be output
- The hash signs `##` after the String are used to remove extra newline characters
- The String, as code, is then assigned to the variable `$greetings`
- Whenever the variable is referenced, the String will be output
- The code associated with a variable by `#define` is executed only when the variable is referenced
- The type of the variable created by `#define` is `org.apache.velocity.runtime.directive.Block$Reference`
- Such an object has a `toString` method
- If the variable references a String, then the variable can be used as a String:

```
#define( $greetings )
Hello, World!##
#end

#set( $str = """$greetings"" is a classic example"
)
$str
```

Using `#define` to Define a Macro

- We can use a `#define` directive to enclose a macro definition:

```
#define( $myMacroExists )
#macro( myMacro )
myMacro invoked
#end
#end

## test if macro is defined
#if( $myMacroExists )
#myMacro
#end
```

- The intention of this code snippet is to invoke a macro only if the macro is defined
- When this snippets stands alone out of any context, this use of `#define` is relatively pointless
- But imagine that the name of the macro to be invoked is resolved only at render time; so we have code of the following type instead:

```
#define( $myMacroExists )
#macro( myMacro )
myMacro invoked
#end
#end
```

```
## the macro name comes from somewhere
#set( $macroName = "myMacro" )

## test if macro is defined
#set( $code = '#if($' + $macroName + 'Exists)' +
'#' + $macroName + '#end' )
#evaluate( $code )
```

Here the macro name is resolved at a later time, perhaps from a global variable, from some other macro, or from user input, and we may want a way to test if the computed macro name corresponds to a real macro that can be invoked.

Using `#define` to Invoke a Macro

```
#define( $macroInvocation )
#myMacro
#end

## output the global variable
#macro( myMacro )
$myVar
#end

#set( $myVar = 3 )
$macroInvocation ##=> 3
#set( $myVar = 12 )
$macroInvocation ##=> 12
```

- The macro `#myMacro` is defined to output the value of a global variable
- The variable `$macroInvocation` is associated with the invocation of `#myMacro`; whenever `$macroInvocation` is referenced, `#myMacro` will be invoked
- The snippet outputs 3 and 12

Subtle Differences between `#define` and `#set`

- The simple "Hello, World!" example above may suggest that `#define` is equivalent to `#set`:

```
#define( $greetings )
Hello, World!##
#end
$greetings
$greetings

#set( $greetings2 = "Hello, World!" )
$greetings2
$greetings2
```

- However, the macro invocation example above clearly indicates

that `#define` is different than `#set`

- Subtle differences between the two:
- The variable in the `#define` structure is associated with a block of code, not necessarily a value
- The type of the variable is `org.apache.velocity.runtime.directive.Block$Reference`
- When the variable is referenced, the associated block of code is executed
- In the simple "Hello, World!" example, the variable `$greetings` is not associated with a String, but rather with a command, so to speak, to output a String
- With the variable `$greetings`, we cannot really manipulate the String `Hello, World!` by calling any `java.lang.String` method, unless the String value is assigned to another variable
- To access the String, we need to call the `toString` method through `$greetings`:

```
#define( $greetings )
Hello, World!##
#end
$greetings                ## => Hello, World!
$greetings.class.name     ## =>
org.apache.velocity.runtime.directive.Block$Reference

#set( $greetingsStr = $greetings.toString() )
$greetingsStr             ## => Hello, World!
$greetingsStr.class.name  ## => java.lang.String
```

- The `toString` method returns whatever String returned from the execution of the code
- Even in `#set( $str = """$greetings"" is a classic example" )`, the `toString` method is called implicitly
- To really see the differences between the two, and what `toString` returns, consider another `#define` structure:

```
#define( $myCommands )
#macro( myMacro $var )
Output from macro: $var
#end

#define( $innerDefine )
innerDefine String
#end

$innerDefine
#end

$myCommands
Output toString: $myCommands.toString()
```

```
#myMacro( 12 )  ## => 12
Output toString: $myCommands.toString()
#myMacro        ## => $var
Output toString: $myCommands.toString()
```

And here is the output:

```
innerDefine String
Output toString: innerDefine String
Output from macro: 12
Output toString: innerDefine String
Output from macro: $var
Output toString: innerDefine String
```

- `#define` is used to define a macro `#myMacro` and an inner `#define` structure; there is no String in this outer structure
- The line `$myCommands` is used to reference the variable to generate the macro and the inner `#define`; the macro is not invoked at this point
- Every time when `$myCommands.toString()` is called, `innerDefine String`is output
- `#myMacro( 12 )` is used to invoke the macro by passing in the value 12, and this outputs `Output from macro: 12`
- `#myMacro` is used to invoke the macro again without passing in any argument, and the local variable `$var` exists but is undefined; therefore, the output of the macro is `Output from macro: $var`
- Therefore, `#define` does not really return any Strings, not even in the simple "Hello, World!" example; rather, `$greetings.toString()` does
- What the `toString` method returns is whatever String or Strings appearing as code within the `#define` structure, and it could be Strings from inner `#define` structures; but it does not return anything from macros defined inside the structure

`#define` and Macros

- Both `#define` and macros are used to create reusable code
- A `#define` structure can contain other `#define` structures or macros
- A macro can contain other macros or `#define` structures
- The major difference between `#define` structures and macros is that macros accepts parameters so that they provide a context for creation of local variables; a `#define` structure, unless it contains macros, only works with global variables
- Another difference between the two, as we saw above, is that Strings inside inner `#define` structures are returned in the `toString` method of the outer variable, whereas macros do not contribute anything to `toString`

Mixing up Global and Local Variables

```
#define( $macroInvocation )
#myMacro( $myVar )
#end

#macro( myMacro $var )
$var
#end

#set( $myVar = 3 )
$macroInvocation
#set( $myVar = 12 )
$macroInvocation
```

- The macro `#myMacro` defines a local variable `$var` and outputs its value
- The `#define` directive associates the variable `$macroInvocation` with an invocation of the macro by passing in a global variable `$myVar` as an argument
- Since the global variable is passed into the macro, referencing the variable `$macroInvocation` is enough to output the value of the global variable
- The macro itself is unaware of the existence of the global variable

Importing a Format on Demand

- When a format containing `#define` is imported, the code of the format is placed inline, yet whatever placed within `#define` is not referenced
- If a `#define` directive is used to wrap around another `#import` directive, this latter `#import` is not executed unless the variable in `#defined` is referenced
- Consider the following code found in `chanw-library-import`:
  ```
  #define( $chanwImportDatabase )
  #import( "site://${chanw_framework_site_name}
  /core/library/velocity/chanw/chanw-database-
  utilities" )
  #end
  ```
- When the format `chanw-library-import` is imported, these three lines of code is also imported
- Yet the format `chanw-database-utilities` is not imported, until the variable `$chanwImportDatabase` is referenced
- Therefore, we can use this mechanism to import a format on demand:
  ```
  #import( "site://_brisk/core/library/velocity
  /chanw/chanw-library-import" )
  ```

```
$chanwImportDatabase

## database code here
```

- Without referencing the variable `$chanwImportDatabase`, the library code related to database is not available

The `#evaluate` Directive

- The `#evaluate` directive can be used to dynamically evaluate a String as code; that is to say, we can formulate a String, to be used as code, by using String methods and operations
- Parts of the String can come from different sources
- We can also use a String buffer to create the String
- The String can contain any code, and `#evaluate` will execute the code at render time
- Before `#evaluate` is invoked, the String is not validated
- Example:

```
#set( $myStatement =
    '#set($myVar=3)' +
    '#macro(myHelloWorld)Hello, World!#end' +
    '$myVar' + $_EscapeTool.N +
    '#myHelloWorld' )
#evaluate( $myStatement )
```

- To force the String interpretation and defer the code interpretation, single quotes are used in the `#set` directive
- The variable `$myStatement` in fact stores the following String:

```
#set($myVar=3)#macro(myHelloWorld)Hello,
World!#end$myVar
#myHelloWorld
```

- When this String is evaluated, the value of the variable `$myVar` is output, and the macro `#myHelloWorld` is invoked

```
3
Hello, World!
```

- Important: you can turn any piece of Velocity code into a String (well, by taking care of quotes and escaping everything necessary), and then pass it into the `#evaluate` directive
- You can use this technique to bypass validation of macro code

Single Quotes vs. Double Quotes

- Unlike `#set`, single quotes and double quotes normally do not make any difference in the context of `#evaluate`:

```
#macro( myMacro )
Hello, World!
#end
```

```
#evaluate( '#myMacro' )
#evaluate( "#myMacro" )
```

Here both `#evaluate` statements output the same message `Hello, World!`

- However, if we want to interpolate a variable in the String to be evaluated, then we have to use double quotes:

```
#macro( myMacro )
Hello, World!
#end

#set( $secretWord = 'Macro' )
#evaluate( '#my$secretWord' )

#evaluate( "#my$secretWord" )
```

The first `#evaluate` statement outputs `#myMacro` as a String, whereas the second `#evaluate` statement invokes the macro and outputs `Hello, World!`

- Also note that since `#evaluate` creates a context for String interpretation, the variable `$secretWord` is interpolated even within single quotes

Using `#define` and `#evaluate` to Implement a Switch of String Values

```
#define( $case1 )
Hello, World!##
#end

#define( $case2 )
Welcome to Velocity!##
#end

#set( $switch = "case2" )
#evaluate( "$$switch" )
```

- Two variables, `$case1` and `$case2`, are associated with two blocks of code by the `#define` directive
- We can put any code inside `#define`
- The variable `$switch` is assigned a name of a variable defined by the `#define` directive
- The `#evaluate` directive is used to turn the value of `$switch` back to a variable and reference it
- Note the use of double quotes and double `$` signs in the `#evaluate` statement: we not only want to interpolate the variable `$switch` (hence the second `$` sign), but also want to turn `$case2` into a String `"$case2"` for `#evaluate` so that the variable `$case2` can be referenced (hence the double quotes and the first `$`

sign)

- Without quotes (`#evaluate( $$switch )`), the first `$` is ignored, and `#evaluate( $switch )` is executed to output the String `case2`
- With single quotes instead (`#evaluate( '$$switch' )`), `$switch` is interpolated with `case2`, but `$case2` is output as a String

Using `#define` and `#evaluate` to Implement a Switch of Integer Values

The previous snippet can be modified slightly to allow for a switch of integer values

```
#define( $case1 )
Hello, World!##
#end

#define( $case2 )
Welcome to Velocity!##
#end

#set( $switch = 1 )
#evaluate( "$case$switch" )
```

Since `#evaluate` creates a context for String interpretation, the `#set` statement can be rewritten as `#set( $switch = "1" )` without making any difference.

Implementing a Switch Using Macros

- A switch structure is just the work of two things together:
- A value (eventually, a String) determining which block of code should be executed; the value can be a name, or part of a name, of a variable, or a name, or part of a name, of a macro
- The blocks of code, each associated with a different value
- It is possible to implement such a mechanism using macros:

```
#macro( sayHello )
Hello, World!
#end

#macro( sayWelcome )
Welcome to Velocity!
#end

#set( $switch = "Hello" )
#evaluate( "#say$switch" )
```

- Here we can use the switch value to recreate the name of the intended macro, and use the `#evaluate` directive to invoke it

- Just like `#define`, we can put any code in such a macro
- Unlike `#define`, macros can take parameters
- Parameters can be set before `#evaluate` is used:

```
#macro( sayHello $msg )
Hello, $msg!
#end

#macro( sayWelcome $msg )
Welcome to $msg!
#end

#set( $switch  = "Welcome" )
#set( $message = "Java" )
#evaluate( "#say$switch('$message')" )
```

- Note the use of quotes in this example: double quotes outside, and single quotes inside
- The outer pair of double quotes force the interpolation of both `$switch` and `$message`; before `#evaluate` is executed, the String value is `#sayWelcome('Java')`
- It is this String, as code, that is being evaluated

The Problem of Passing Variables by Reference

- When we pass a variable as an argument into a macro, what is passed in is the value of the variable, not its reference
- That means that we cannot modified the variable itself inside the macro, unless the variable is a global variable, and the macro is aware of its existence
- Example:

```
#set( $myVar = 3 )
#doubleIt( $myVar )
$myVar ##=> 3, not 6
$var   ##=> 6

#macro( doubleIt $var )
    #set( $var = $var * 2 )
#end
```

The intention here is to modify the local variable `$var` so that its value is doubled. But the code does not work. That is to say, the global variable `$myVar` is not affected by the macro.

- The problem: what is visible inside the macro is the value 3, not the variable `$myVar`, and when `#set( $var = $var * 2 )` is executed, the second instance of `$var` is replaced by the value 3, whereas the first instance is used to create a new global variable named `$var`
- The variable `$myVar` is not visible in the macro

- What the macro does is to use the value passed, double it, and assign the new value to a new global variable `$var`
- Somehow, we need a mechanism to make the variable visible inside the macro to solve this problem

Solving the Problem of Passing Variables by Reference

- Assuming that a macro is not aware of the existence of a global variable, the information of the global variable must be passed into the macro
- Passing in the value of the global variable is not helpful at all; what we need is to pass in the variable itself, not its value
- We can pass in the name of the variable, and use `#evaluate` to reconstruct the variable itself
- Example:

```
#set( $myVar = 3 )
#doubleIt( "myVar" )
$myVar

#macro( doubleIt $varName )
    #set( $stmt = '#set($' + $varName + "=$" +
$varName + "*2)" )
    #evaluate( $stmt )
#end
```

- After the `#set` statement is executed, `$stmt` stores the String `#set($myVar=$myVar*2)`; note how the global variable, as code, is reconstructed from the name passed in
- When the `#evaluate` statement is executed, the global variable "becomes" available to the macro
- Lesson learned: names of macros, variables, and even Velocity code, can be passed around as Strings from macro to macro, and `#evaluate` enables us to turn these Strings back to code
- This is an extremely powerful technique available only in Velocity but not in XSLT

A Slightly Different Implementation Using `#define` and `#evaluate`

- Consider the following:

```
#set( $myVar = 3 )
#doubleIt( "myVar" )
$myVar

#macro( doubleIt $varName )
    #define( $code )
        #set( $stmt = '#set' +
```

```
"($$varName=$$varName*2)" )
        #evaluate( $stmt )
    #end
    $code ## reference the variable
#end
```

- The `#define` directive is used inside the macro so that the variable `$varName` can be made visible inside `#define`
- But note that the variable `$code` must be referenced inside the macro
- If the variable is not referenced within the macro, the value of `$varName` becomes unavailable to `#define`
- That is to say, the scope of `#define` can shield the local variable of the macro, unless the variable associated with `#define` is referenced when the value of the local variable is still available
- This illustrates nicely the limitation of `#define`: because it is impossible to pass a parameter value into `#define`, any value used by `#define` must be made available outside the immediate scope of `#define`:

```
#set( $myVar = 3 )
#doubleIt( "myVar" )

#macro( doubleIt $varName )
    #define( $code )
        #set( $stmt = '#set' +
"($$varName=$$varName*2)" )
        $stmt ##=> #set($$varName=$$varName*2)
        #evaluate( $stmt )
    #end
#end

## reference the variable here
$code
```

In this example, the variable `$varName` is not resolved in `#define` because the variable `$code` is not referenced when the value of `$varName` is still available; hence the code is invalid.

A Library Macro `#chanwSetVariable` to Set Variables

- Here is a library macro that can be used to set variables:

```
#macro( chanwSetVariable $chanwSetVariableVar
$chanwSetVariableVal $chanwSetVariableEscape )
    ## check the variable name
    #if( !$chanwSetVariableVar ||
$chanwSetVariableVar.class.name !=
"java.lang.String" || $chanwSetVariableVar == "" )
#chanwOutputErrorMsg( $CHANW_LIBRARY_FOLDER_PATH
```

```
$chanw_framework_site_name
$CHANW_INITIALIZATION_FORMAT_NAME "The variable
name $chanwSetVariableVar is not acceptable."
"ln;4V84GBIUBJKLV" )
        #stop
    #end
##
    ## escape defaulted to true
    #if( !$chanwSetVariableEscape.Class.Name ||
$chanwSetVariableEscape == true )
        #set( $chanwEscapeValue = true )
    #else
        #set( $chanwEscapeValue = false )
    #end
##
    #if( !$chanwSetVariableVal.Class.Name )
        #set( $chanwValue = "" )## default value
    #elseif( $chanwSetVariableVal.class.name ==
"java.lang.String" )
        #if( $chanwEscapeValue )
            #set( $chanwValue = $_EscapeTool.xml(
$chanwSetVariableVal ).trim() )
        #else
            #set( $chanwValue =
$chanwSetVariableVal.trim() )
        #end
    ## this is an object
    #else
        #set( $chanwValue = 'chanwSetVariableVal'
)
    #end
##
    #if( !$chanwValue.equals(
'chanwSetVariableVal' ) )
        #set( $chanwSetVarValStatement = '#' +
"set(" + "$" + "$chanwSetVariableVar =
'$chanwValue')" )
    #else
        #set( $chanwSetVarValStatement = '#' +
"set(" + "$" + "$chanwSetVariableVar = $" +
$chanwValue + ")" )## object assignment
    #end
##
    #evaluate( $chanwSetVarValStatement )
#end
```

- First, we want to make sure that the `$chanwSetVariableVar` variable points to a String object
- Next, if the `$chanwSetVariableVal` variable is undefined, we will

use the empty String as the default value

- We create a statement to set the global variable by assigning the value to it
- Lastly, we use `#evaluate` to perform the assignment
- Note how quotes and interpolation are used to create the statement
- Code example:
  `#chanwSetVariable( "myVar" 3 )`

  This line is equivalent to `#set( $myVar = 3 )`. The difference between the two is that `#chanwSetVariable` can be used to set a variable whose name is unknown when the library macro was written.

Another Library Macro Using `#chanwSetVariable`

- Here is another library macro using `#chanwSetVariable`:

```
#macro( chanwSetVariableToNonEmptyString
$chanwSetVariableToNonEmptyStringVar
$chanwSetVariableToNonEmptyStringList )
#**##if(
$chanwSetVariableToNonEmptyStringList.class.name
!= $JAVA_UTIL_ARRAY_LIST_CLASS_NAME )
#chanwOutputErrorMsg( $CHANW_LIBRARY_FOLDER_PATH
$chanw_framework_site_name
$CHANW_INITIALIZATION_FORMAT_NAME "A list of
strings is required." "3nipv4knvjkn" )
        #stop
#**##end
##
#**##set( $chanwSize =
$chanwSetVariableToNonEmptyStringList.size() )
##
#**##if( $chanwSize == 0 )
#chanwOutputErrorMsg( $CHANW_LIBRARY_FOLDER_PATH
$chanw_framework_site_name
$CHANW_INITIALIZATION_FORMAT_NAME "A list of
strings is required." "MLRVIOUGOJKNY" )
        #stop
#**##end
##
#**##set( $chanwValue = "" ) ## empty string, the
last resort
##
#foreach( $num in [ 1..$chanwSize ] )
#set( $chanwIndex = $num - 1 )
#if( $chanwSetVariableToNonEmptyStringList[
$chanwIndex ].class.name !=
$JAVA_LANG_STRING_CLASS_NAME )
```

```
#chanwOutputErrorMsg( $CHANW_LIBRARY_FOLDER_PATH
$chanw_framework_site_name
$CHANW_INITIALIZATION_FORMAT_NAME "Not a string."
"bijog5iojnbm.gl" )
#stop
#end
#if( $chanwSetVariableToNonEmptyStringList[
$chanwIndex ].trim() != "" )
#set( $chanwValue =
$chanwSetVariableToNonEmptyStringList[ $chanwIndex
].trim() )
#break
#end
#end
##
#chanwSetVariable(
$chanwSetVariableToNonEmptyStringVar $chanwValue )
#end
```

- This macro accepts a variable name and a list of Strings as parameters
- The macro will look for the first non-empty String and assign it to the variable
- If all Strings in the list are empty, then the variable is assigned the empty String
- This macro invokes `chanwSetVariable` to assign the intended value to the variable
- Code example:
  `#chanwSetVariableToNonEmptyString( "myVar" [ " ", "", "content", "More content" ] )`

  After this line is executed, the variable `$myVar` will store `content`
- Also see drulykgSetVar by German Drulyk

`#chanwReviveGlobalVariable`

- The point of using a macro to set a variable: when the variable name is embedded with another variable:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )

#set( $namespace = "chanw" )
#set( $myVarName = "MyVar" )
#chanwSetVariable( "$namespace$myVarName" 3 )

$chanwMyVar
```

  Here the variable name `$chanwMyVar` is computed, with a prefix from another variable `$namespace`, possibly passed in from the client, and a second part from yet another variable.

- Often we need to work with names that are passed around from macro to macro and from format to format, and when writing reusable code, these names are unknown
- But then, the last line of the code snippet (i.e., `$chanwMyVar`) becomes impossible if the name of the variable is computed; right now it is hard-coded in the snippet
- The question is this: we have a way to set a variable, whose name is resolved after the library code is built; but how do we refer to this variable in the library code if we don't know its name?
- Note that even though the name is unknown when we write the library code, it must still be resolved when the library code is put into use
- That is to say, we can rely on the String value, i.e., the variable name, to be made available in the client code
- With the variable name, we can recover the value from the global variable bearing that name
- Here is a macro that does that:

```
#macro( chanwReviveGlobalVariable
$chanwReviveGlobalVariableGlobalVarName
$chanwReviveGlobalVariableNs )
    #set( $chanwReviveGlobalVariable = "" )
    #set( $chanwGlobalVarName =
$chanwReviveGlobalVariableGlobalVarName )
##
    #if( $chanwReviveGlobalVariableNs.Class.Name
== $JAVA_LANG_STRING_CLASS_NAME &&
$chanwReviveGlobalVariableNs != "" )
        #set( $chanwGlobalVarName =
$chanwReviveGlobalVariableNs +
$chanwReviveGlobalVariableGlobalVarName )
    #end
##
    #set( $chanwStmt = '#' +
'set($chanwReviveGlobalVariable=' +
"$$chanwGlobalVarName" + ')' )
    #evaluate( $chanwStmt )
#end
```

- The name of the global variable is passed into the library macro
- Since the name of the global variable is unknown to the library, I use the variable `$chanwReviveGlobalVariable` to stand for the unknown global variable
- The value associated with the unknown global variable is assigned to `$chanwReviveGlobalVariable`
- That is to say, even though we do not know the name of the global variable when writing the library code, the name will be made

available when the library code is used; and when the name of the variable is supplied, we can get its value and assign it to `$chanwReviveGlobalVariable`

- Here is code to use the macro:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )

#set( $namespace = "chanw" )
#set( $myVarName = "MyVar" )
#chanwSetVariable( "$namespace$myVarName" 3 )

#chanwReviveGlobalVariable( $myVarName $namespace
)
$chanwReviveGlobalVariable
```

Note that in this code snippet, the name `chanwMyVar` never appears.

#chanwInvokeMacro

- Besides variable names, macro names can also be embedded with variables
- The library macro `#chanwInvokeMacro` can be used to invoke any macro (the targeted macro)
- When invoking `#chanwInvokeMacro`, the name of the targeted macro must be passed in as a String
- The simplest way to use this macro:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )

#macro( myMacro )
    myMacro invoked
#end

#chanwInvokeMacro( "myMacro" )
```

- If the targeted macro accepts parameters, then we need a second parameter for `#chanwInvokeMacro`, a list of variable names (names of those variables that need to be passed into the targeted macro):

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )

#macro( myMacro $p1 $p2 )
    myMacro invoked with $p1 $p2
#end

#set( $param1 = "Hello," )
#set( $param2 = "World!" )
```

```
#chanwInvokeMacro( "myMacro" [ "param1", "param2"
] )
```

- Here is the code of `#chanwInvokeMacro`:

```
#macro( chanwInvokeMacro
$chanwInvokeMacroMacroName
$chanwInvokeMacroMacroParams )
#**##if( $chanwInvokeMacroMacroName.Class.Name ==
$JAVA_LANG_STRING_CLASS_NAME )
#*    *##set( $chanwStmt =
"#$chanwInvokeMacroMacroName(" )
##
#*    *##if(
$chanwInvokeMacroMacroParams.Class.Name ==
$JAVA_UTIL_ARRAY_LIST_CLASS_NAME )
#*        *##if(
$chanwInvokeMacroMacroParams.size() > 0 )
#*            *##foreach( $macroParam in
$chanwInvokeMacroMacroParams )
#*                *##if( $macroParam.Class.Name ==
$JAVA_LANG_STRING_CLASS_NAME )
#*                    *##set( $chanwStmt =
$chanwStmt + "$" + $macroParam + " " )
#*                *##end
#*            *##end
#*        *##end
#*    *##end
##
#*    *##set( $chanwStmt = $chanwStmt.trim() )
#*    *##set( $chanwStmt = $chanwStmt + ")" )
#*    *##evaluate( $chanwStmt )
#**##end
#end
```

- What's the point: `#chanwInvokeMacro` allows us to invoke a macro whose name is computed and possibly embedded with other variables
- With this macro, we can start thinking about automatic macro invocation, a mechanism deployed in the Standard Model, to deal with blocks attached to pages
- The crux of automatic macro invocation is this: for any given block of any type, we can derive a name, and from this name, we compute the name of the targeted macro designed to process the block, and use `#chanwInvokeMacro` to invoke it to process the block

How to Work with `#evaluate`

- `#evaluate` creates a context for String interpretation

- Characters, including quotes, the hash signs, and the dollar signs, used in the code/String to be evaluated must be properly escaped
- Quotes must be nested properly
- Using String concatenation, mixed with all escaped character sequences, can be confusing at times when working with long code
- Let us create a macro named `#chanwInvokeMacroIfExists` that can be invoked to invoke another macro (the targeted macro) by invoking `#chanwInvokeMacro`
- The specification of `#chanwInvokeMacroIfExists`:
- When `#chanwInvokeMacroIfExists` is invoked, the name of the targeted macro, `${macroName}` must be passed in as a String
- If the targeted macro accepts one or more parameters, then `#chanwInvokeMacroIfExists` takes a second parameter, a list of variable names to be passed into the targeted macro
- The targeted macro can be defined with `#define`, so that a variable with the suffix `Exists` (like `myMacroExists`) is created around the definition of the targeted macro
- `#chanwInvokeMacroIfExists` checks if the variable `$${macroName}Exists` (where `${macroName}` is the name of the targeted macro) is defined
- If the variable is defined, then pass the macro name `#${macroName}` with all the parameters in to `#chanwInvokeMacro`
- To invoke `#chanwInvokeMacroIfExists`, we need code like the following:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )

#define( $myMacroExists )
    #macro( myMacro $p1 $p2 )
        myMacro invoked with $p1 and $p2
    #end
#end

#set( $param1 = "Hello, " )
#set( $param2 = "World!" )
#chanwInvokeMacroIfExists( "myMacro" [ "param1",
"param2" ] )
```

- The first step is to write the macro `#chanwInvokeMacroIfExists`, containing a String to be evaluated, as a normal macro in a test format, and write out the String as code:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )
```

```
#define( $myMacroExists )
    #macro( myMacro $p1 $p2 )
        myMacro invoked with $p1 and $p2
    #end
#end

#set( $param1 = "Hello," )
#set( $param2 = "World!" )
#chanwInvokeMacroIfExists( "myMacro" [ "param1",
"param2" ] )

#macro( chanwInvokeMacroIfExists $macroName
$paramList )
    #if( $myMacroExists )
        chanwInvokeMacroIfExists invoked
        ## invoke the macro
        #chanwInvokeMacro( $macroName $paramList )
    #end
#end
```

Note that the variable `$myMacroExists` is hard-coded in `#chanwInvokeMacroIfExists` so that we can test the macro.

- `$paramList` is an `ArrayList` object; we need to turn it into a String like `[ "param1", "param2" ]`:

```
#set( $paramList = [ "param1", "param2" ] )

#set( $chanwParamString = "[" )
##
#foreach( $varName in $paramList  )
    #set( $chanwParamString = $chanwParamString +
""" + $varName.toString() + """ )
    #if( $foreach.count < $paramList.size() )
        #set( $chanwParamString =
$chanwParamString + "," )
    #end
#end
#set( $chanwParamString = $chanwParamString + "]"
)
$chanwParamString
```

- Once we have the parameters as a String, we create a String that looks like the invocation of `#chanwInvokeMacro`:

```
    #set( $chanwStmt = "#if( $myMacroExists )
        #chanwInvokeMacro( $macroName
$chanwParamString )
    #end" )
```

- Replace the hard-coded names with variables
- Replace all instances of `$` that should appear in the code with `${DOLLAR}`

- Replace all instances of # that should appear in the code with `${SINGLE_HASH}`
- Put `""` around String variables
- Keep the `$` characters for variables to be interpolated
- Remove all newline characters
- We will have the following:

```
#set( $chanwStmt =
"${SINGLE_HASH}if(${DOLLAR}${macroName}Exists)${SINGLE_HASH}chanwInvokeMac
$chanwParamString)${SINGLE_HASH}end" )
$chanwStmt
```

  The variable `$chanwStmt` should store the String `#if($myMacroExists)#chanwInvokeMacro("myMacro" ["param1","param2"])#end`.

- Make sure this is the code we want
- The last step is to evaluate the String
- Here is the code of `#chanwInvokeMacroIfExists`:

```
#macro( chanwInvokeMacroIfExists
$chanwInvokeMacroIfExistsMacroName
$chanwInvokeMacroIfExistsListOfParams )
    #if(
!$chanwInvokeMacroIfExistsListOfParams.Class.Name
)
        #set( $chanwParamString = "" )
    #elseif(
$chanwInvokeMacroIfExistsListOfParams.Class.Name
== $JAVA_UTIL_ARRAY_LIST_CLASS_NAME &&
$chanwInvokeMacroIfExistsListOfParams.size() > 0 )
        #set( $chanwParamString = "[" )
##
        #foreach( $varName in
$chanwInvokeMacroIfExistsListOfParams  )
            #set( $chanwParamString =
$chanwParamString + """ + $varName.toString() +
""" )
            #if( $foreach.count <
$chanwInvokeMacroIfExistsListOfParams.size() )
                #set( $chanwParamString =
$chanwParamString + "," )
            #end
        #end
        #set( $chanwParamString =
$chanwParamString + "]" )
    #else
        #set( $chanwParamString = "[]" )
    #end
##
```

```
    #set( $chanwStmt =
"${SINGLE_HASH}if(${DOLLAR}${macroName}Exists)${SINGLE_HASH}chanwInvokeMa
$chanwParamString)${SINGLE_HASH}end" )
    #evaluate( $chanwStmt )
#end
```

What about &&, < and >?

- Characters or character sequences like `&&`, < and > can appear in Velocity code but not in Strings
- When creating a String to be evaluated, these characters must be escaped
- I used `-aa-`, `-lt-` and `-gt-` to replace these characters when creating the String
- Right before evaluating the String, we need to use `java.lang.String.replaceAll` and turn the String back into code:

```
#set( $chanwCode = $code.replaceAll( "-aa-", "&&"
).replaceAll( "-lt-", "<" ).replaceAll( "-gt-",
">" ) )
#evaluate( $chanwCode )
```

Differences between XSLT and Velocity Again

- It should be obvious now what roles global variables and `#set` can play in Velocity
- Powerful reusable Velocity code can be written using `#evaluate` by turning Strings into code
- Java reflection is available to Velocity but only to a limited extent to XSLT
- Combining all these facts, the powerful Upstate Velocity library can only be written in Velocity; this is absolutely impossible with XSLT
- From this point on, the following tasks can only be done with Velocity:
- Creating global utility objects
- Creating a database connectivity so that the Cascade database can be accessed
- Reusing XSLT formats in Velocity
- Making web services available to Velocity and processing the returned results