upstate.edu

Lesson 11: #import, #break, #stop, and Debugging

13-17 minutes

Learning Objectives

- Learn how to:
- Use the `#import` directive and library macros to import formats efficiently
- Import formats on demand
- Use the `#break` directive to group functional code
- Use the `#break` and `#stop` directives for debugging purposes
- Debug large formats

Collapse all

Organizing Macros

- To promote reusability, blocks of reusable code should be written as macros
- Try to keep macros small for easy maintenance: divide-and-conquer
- A macro should have a single functionality
- Macros having multiple functionality (i.e., doing too much) should be split into multiple macros
- Macros can invoke other macros for better division of labor
- Macros can contain other macros
- Possible ways of organizing macros in formats:

- Order macros alphabetically with their names
- Define formats with functionality and group macros with similar functionality in the same format
- When necessary, put a single macro with unique functionality in a format
- Organize formats in designated folders
- When dealing with presentational macros (groups of macros used to render the same set of data for different designs), macros can be defined within a macro, so that macros with close dependencies can be put together

Cyclic Import

- When building reusable formats, one format may import another format
- A format can import itself through other formats
- Cyclic import occurs when the path of importing formats is a circle
- Examples:
- When a format imports itself explicitly
- When format A imports format B, which imports format A
- The chain of import goes back to the original format: A -> B -> ... -> N -> A
- Cyclic import can introduce bugs that are extremely hard to identify and fix
- This happens when reusable macros (of the same name) are redefined in different imported formats, or when global variables are defined in some formats in the cyclic path but not in others
- Cyclic import can be introduced when the `#import` directive is used within a macro
- Good indication of wrong placement of macros: if a macro has to import a format to work, then probably the macro should be placed in that format

Importing All Formats in a Folder

- We should organize all reusable formats in a designated folder
- Assuming that the folder does not contain sub-folders, we can import all formats in the folder using code of the following type:

```
#foreach( $item in $_.locateFolder( $folderPath,
$siteName ).Children )
    #if( $item.AssetType == 'format' )
        #drulykgImportPassThrough(
"site://${siteName}/${item.Path}" )
    #end
#end
```

- If we want to make sure only Velocity formats are imported, we can add one more condition:

```
    #if( $item.AssetType == 'format' &&
$item.Class.Name ==
$COM_CASCADE_SCRIPT_FORMAT_CLASS_NAME )
```

- If the designated folder contains sub-folders, then we should define a recursive macro to import all formats in that folder

Importing Formats on Demand

- `#import` directives can be grouped together in a single format, which serves the purpose of being the entry point to a library
- When using the library, just import this single format
- However, we may want a way to facilitate conditional import, so that some formats can be imported on demand
- For example, a format designed to deal with the Cascade database should not be imported if it is not needed or if the users are not sophisticated enough to handle the database
- We can use the `#define` directive, which will be introduced in the intermediate course, to import formats on demand
- Roughly, `#define` associates a variable name with a block of code

- We can wrap an `#import` directive inside a `#define` directive like this:
```
#define( $chanwImportDatabase )
#import( "site://${chanw_framework_site_name}
/core/library/velocity/chanw/chanw-database-
utilities" )
#end
```
- When the format, for example, `chanw-library-import` is used as the entry point, we can import the library by using the following:
```
#import( "site://${chanw_framework_site_name}
/core/library/velocity/chanw/chanw-chanw-library-
import" )
```
- Although the entry point to library is imported, the format `chanw-database-utilities` is not
- To import `chanw-database-utilities` as well, we need an extra line:
```
#import( "site://${chanw_framework_site_name}
/core/library/velocity/chanw/chanw-library-import"
)
$chanwImportDatabase
```
- Note that without `#import( "site://${chanw_framework_site_name}/core/library /velocity/chanw/chanw-library-import" )`, the variable `$chanwImportDatabase` is undefined

Using `#break`

- The `#break` directive can be used to break out of the execution of a block of code
- For example, when it is used inside embedded `#foreach` structures, it can break out of any level of the looping structure
- When it is used in a macro, it can lead to the exit of the macro (similar to a `return` statement in programming)

- When it is used at the top level of a format (not inside any scoped block), it can terminate the execution of the format
- When dealing with a large format (e.g., the format attached to DEFAULT), one way to group lines of code of distinct functions is to put the code in macros and to invoke macros at the right places
- Another way to group lines of code is to wrap the code within a conditional (i.e., `#if`…`#end`), with the `#break` directive at the very end right before `#end`:

```
#if( $condition )
    ## code
    #break
#end
```

- This use of `#break` enables the same format to perform several distinct functions without defining any macros
- The same logic can be applied to a macro, which consists of a group of distinct but related functionality

Using `#stop`

- The `#stop` directive can be used to stop the execution of Velocity code altogether
- When both `#break` and `#stop` are used at the top level of a format, they can have similar effect
- However, there can be some subtle difference between the two
- Consider the following format named `break-stop`:

```
#set( $one = "one" )
#set( $two = "two" )
#set( $three = "three" )

#if( $break )
    #break
#end
```

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

#if( $stop )
    #stop
#end

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

- Now consider a format named `test-break-stop` that imports `break-stop`:
  ```
  #import( "formats/tutorial/introductory/9/break-stop" )
  $five
  ```
- Since the variable `$five` is defined in `break-stop`, `test-break-stop` will output the string "five"
- Now we set the variable `$break` to true before the import:
  ```
  #set( $break = true )
  #import( "formats/tutorial/introductory/9/break-stop" )
  $five
  ```
  The variable `$five` is not defined because the definition appears after the conditional `#break` directive and the condition is met
- The output of `test-break-stop` is "$five"
- If we have the following:
  ```
  #set( $stop = true )
  #import( "formats/tutorial/introductory/9/break-stop" )
  $five
  ```
  Then `test-break-stop` outputs nothing
- This is because when `break-stop` is imported, the execution is terminated altogether, and it never comes back to the line `$five`
- Important: When the directive `#stop` is evaluated (`#evaluate('#stop')`, the execution of the code will be scoped

within the string to be evaluated; the code after `#evaluate` will still be executed

Using `#chanwOutputErrorMsg`

- There is macro named `#chanwOutputErrorMsg` that you can use to output meaningful debugging messages from a reusable format
- The macro takes five string parameters:
- `$folderPath`: the path of the parent folder containing the format
- `$siteName`: the name of the site containing the format
- `$formatName`: the name of the format
- `$msg`: the message to be output
- `$key`: a unique random string containing any characters; each call to `#chanwOutputErrorMsg` must be associated with a unique key string
- Example: the reusable format named `output-message` contains the following:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )
#set( $folderPath = "formats/tutorial
/introductory/9" )
#set( $siteName   = "_chan_core" )
#set( $formatName = "output-message" )

#set( $one = "one" )
#set( $two = "two" )
#set( $three = "three" )

#set( $four = "four" )
#chanwOutputErrorMsg( $folderPath $siteName
$formatName "Output message 1." "ICENEJiern" )

#set( $five = "five" )
#chanwOutputErrorMsg( $folderPath $siteName
```

```
$formatName "Output message 2."
"iojoigo4jri;jenrgn" )
```

- Note that the two invocations of `#chanwOutputErrorMsg` occur in line 11 and line 14
- The following format named `testoutput-message` contains the following:

  ```
  #import( "formats/tutorial/introductory/9/output-
  message" )

  $five
  ```

  And this is the output:

  ```
  formats/tutorial/introductory/9/output-message,
  line 11: Output message 1.

  formats/tutorial/introductory/9/output-message,
  line 14: Output message 2.

  five
  ```

- That is to say, the macro outputs all information related to the source of a message, including the line number where the message is output

Debugging Large Formats Using Binary Search

- We should try to avoid having lengthy code in formats and macros
- But sometimes it is just impossible to void that without unnecessarily defining extra macros
- Let us assume that we are working with a macro containing 200 lines of code
- We know that there is something wrong in the macro, and there could be bugs within; we just don't know where the bugs are
- One of the governing principles in programming, i.e., divide and conquer, will help

- With 200 lines of code to debug, the job can be overwhelming if we are to walk down the code line by line
- But what if we just need to look at the first half?
- If there is still something wrong within the first half, how about just the first quarter?
- By constantly cutting the code we want to examine and debug by half, we will eventually, after a few trials, get down to one or two lines of code
- We focus on the one or two lines of code and fix bugs within
- We then backtrack to the last dividing, working with three or four lines of code
- This is called binary search
- If there is nothing wrong with the first half, we look at the second half and perform binary search on the second half until we fix all bugs

Setting up Breakpoints

- When performing binary search to debug lengthy code, it is good to have break points within the code so that we can output something, like printing the value of a variable
- We have seen how to use the macro named `#chanwOutputErrorMsg`
- With a little bit of extra code and the `#break` or `#stop` directives, we can stop the execution of the code at a desirable point
- Let us look at how to set up breakpoints; here is a few lines of code that we can add to the top of a format we want to debug:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )
#set( $point = 2 )
#evaluate(
"${SINGLE_HASH}define(${DOLLAR}breakAt${point})${SINGLE_HASH
)
```

- We will look at `#evaluate` and `#define` in the intermediate course
- In effect, this is just a simple `switch…case` structure
- We can use these three lines of code to the top of a format without worrying what they do
- The variable `$point` can be assigned a numeric value like 1, 2, or 3
- Within the code we want to debug, we can add code of the following type:

```
#import( "site://_brisk/core/library/velocity
/chanw/chanw-library-import" )
#set( $point = 2 ) ## set to a point we want to
stop
#evaluate(
"${SINGLE_HASH}define(${DOLLAR}breakAt${point})${SINGLE_HASH
)

## code to skip

#if( $breakAt1 )
    ## output a message or the value of a variable
here
    ## or invoke #chanwOutputErrorMsg
    #stop
#end

## more code here

#if( $breakAt2 )
    ## output a message or the value of a variable
here
    ## or invoke #chanwOutputErrorMsg
    #stop
#end
```

…

```
#if( $breakAt7 )
    ## output a message or the value of a variable
here
    ## or invoke #chanwOutputErrorMsg
    #stop
#end
```

- Now we can assign a numeric value to `$point` matching the position where we want to stop

Commenting out a Block of Code

- While we are working on a lengthy format, besides setting up breakpoints, we may want to comment out large chunk of code so that we don't need to break so often
- It will be tedious to add ## to every line
- The block of code may contains multi-line comments; therefore we cannot use #**# either
- There are a few ways to make blocks of code non-executable (or otherwise)
- The first way to wrap the code with a conditional:

```
#if( false )
    ## code to be skipped
#end
```

- Since the code is wrapped inside `#if( false )`, it will be skipped
- If we want this block of code to be executed, we need to change `false` to `true`
- The second way is to turn the code into a macro:

```
#macro( a )
    ## code to be skipped
#end
```

- The advantage of using a macro is that we can put `#a` right above the code:

```
#a
#macro( a )
    ## code to be skipped
#end
```

- If we want to have the code skipped, we just need to comment out `#a`
- This mechanism will still work even when we are debugging a macro, because macros are allowed within a macro
- The third way is to wrap the code with `#define`:

```
#set( $one = "one" )
#set( $two = "two" )

## skip this part
#define( $a )
#set( $three = "three" )
#set( $four = "four" )
#end

## or not
$a
#set( $five = "five" )
$one
$three
```

A Few Debugging Tricks

- `$x.Class.Name` is your friend: whenever you are dealing with a variable, and you are not sure what that variable is, output its type
- If the variable is not defined, then you will see output of the code
- Output the value of a variable
- Make sure that local variables and global variables have different names

- Make sure that macros are not redefined (or overwritten); that is to say, each macro available in the context must have a unique name
- When working with a list, use `#foreach` whenever possible
- Make sure that global variables are properly reinitialized in loops and macros
- When selecting a single node, use `$_XPathTool.selectNodes( $node, $path )[ 0 ]` and `$node.getChildren( $name )[ 0 ]` instead of `$_XPathTool.selectSingleNode( $node, $path )` and `$node.getChild( $name )`
- When necessary, set up some breakpoints in you code and use `#stop` sensibly
- When necessary, "comment out" large chunks of code and focus on a few lines
- As the last resort, if you want to remove a large chunk of working code from a format that you want to debug, move the code into a new format, and in its original source, add an `#import` directive (out of sight, out of mind); you can move the code back later

---

Examples

- introductory/11_import_break_stop_debugging