UPSTATE
UNIVERSITY HOSPITAL

MENU

Lesson 4: Conditionals and Loops

Learning Objectives

- Understand:
  - What boolean expressions are
  - How an undefined variable is resolved in a boolean context
  - The importance of reinitializing variables in a loop
- Learn how to:
  - Use `#if` and related structures to perform tests
  - Use `#foreach` to loop through items in a list

Collapse all

Conditionals

- A conditional is used to perform some test(s)
- There are four types of conditionals:
  - `#if…#end`
  - `#if…#else…#end`
  - `#if…#elseif…#end`
  - `#if…#elseif…#else…#end`
- There can be, and must be, only one `#if` directive in a conditional, and it must be the first part of the conditional
- There can be at most one `#else` directive in a conditional, and it is optional; if there is one, it must be the last part before `#end`
- The `#elseif` directive is optional, and there can be any number of them in a conditional; if there are any `#elseif` directives, they must appear between `#if` and `#else` (if there is one) or `#end`

- The `#if` and `#elseif` directives are always followed by a pair of parentheses, within which there must be a boolean expression; hence we say that these two directives create a boolean context
- The boolean expression must be evaluated to either true or false
- Both keywords `true` and `false` can appear in the boolean context
- Following the ending parenthesis, before the next directive, code can be provided
- At most only one block of code will be executed within a conditional, depending on which boolean expression is evaluated to true first
- Boolean expressions are evaluated top-down, starting from the one associated with `#if`
- When the first boolean expression is evaluated to true, the associated block of code will be executed, and the execution will jump to `#end` ; if none of the boolean expressions are true, then the entire conditional will be skipped
- An undefined variable, when appearing in the boolean context, is evaluated to false
- Note that Velocity does not provide a `switch` structure

Boolean Expressions

- A boolean expression is an expression appearing in the pair of parentheses immediately following either `#if` or `#elseif`
- A boolean expression is always evaluated to either true or false
- A boolean can be just the keyword `true` or `false` :

```
#if( true )
Voila!
#end

#if( false )
Never here!
#else
Always here.
#end
```

- A boolean expression can be a String:

```
#if( "Hello" )
A String is true!
#else
A String is false! ## this is output
#end
```

Here a String, no matter what it contains (including code to invoke a macro), is always evaluated to false. In fact, all literal values, including lists and maps, are evaluated to false:

```
#if( 3 )
An integer is true!
#else
An integer is false! ## this is output
#end

#if( [ 1..4 ] )
A literal list is true!
#else
A literal list is false! ## this is output
#end
```

- A boolean expression can be a variable:

```
#if( $condition )
An undefined variable is true!
#else
An undefined variable is false! ## this is output
#end

#set( $condition = "" )

#if( $condition )
A defined variable is true! ## this is output
#else
A defined variable is false!
#end
```

- If the variable is undefined, then as a boolean expression, it is evaluated to false
- When the variable is defined with a valid value (a non- `null` value), then as a boolean expression, it is evaluated to true, unless the value is `false`
- Note that when the empty String is assigned to the variable, which is then put in the boolean context, the variable is evaluated to true, not false; yet the empty String in the boolean context is evaluated to false. This again means that a variable is not equivalent to the value it points to.
- A boolean expression can be a boolean statement, containing relational and/or logical operator(s), that is evaluated to either true or false:

```
#set( $children = $contentRoot.getChildren( "system-page" ) )

#if( $children.size() > 0 )
## process system-page elements
#else
Nothing to process.
#end
```

- Since an undefined variable is evaluated to false, we can use a conditional to test if an element in fact exists:

```
#set( $firstPage = $contentRoot.getChild( "system-page" ) )

#if( $firstPage )
process first system-page element
#end
```

If there are no `system-page` children in `$contentRoot`, then `$firstPage` can be undefined, and the conditional will be skipped.

- In fact, a boolean expression can be a direct Java method call:

```
#if( $contentRoot.getChild( "system-page" ) )
process first system-page element
#end
```

Though in this case the element is not stored in a variable, and we may need to call `getChild` again to retrieve the element.

- When the calling object is of type `org.jdom.Element`, `getChildren` always returns a list; when such a list is assigned to a variable, the variable is always evaluated to true, even when the list is empty:

```
#set( $children = $contentRoot.getChildren( "system-data-definition" ) )

#if( $children )
Always true.
#end
```

Therefore, instead of testing the variable, test the size of the list:

```
#set( $children = $contentRoot.getChildren( "system-data-definition" ) )

#if( $children.size() > 0 )
## proceed
#end
```

- That is to say, when `getChild` is used, we can test the variable; but when `getChildren` is used, we should always test the size of the returned list
- The same rule applies to `$_XPathTool.selectSingleNode` vs. `$_XPathTool.selectNodes`

Equality vs. Assignment

- When comparing Java objects for equality, we need to use the equal operator `==`, not the assignment operator `=`:

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

#if( $str == "Hello" )  ## comparison for equality
Identical
#end
```

- An assignment operator used in a boolean context will cause an error
- Besides the equal operator, we can also call the `equals` method (inherited from `java.lang.Object`):

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

#if( $str.equals( "Hello" ) )
Identical
#end
```

List Literals

- A list can be created by using the list notation `[]`:

```
#set( $states = [ "NY", "NJ", "WA" ] )
#set( $fiveToTen = [ 5..10 ] )  ## [ 5, 6, 7, 8, 9, 10 ]
```

- The type of such a list is `java.util.ArrayList<E>`, which is subclass of `java.util.AbstractCollection<E>`
- If we need an array, call the `java.util.ArrayList.toArray` method
- To add an item to a list, use the `add` method
- Since the `add` method returns a boolean value, if we don't want the returned value to be output, we need to assign it to a variable:

```
#set( $myList = [] )
#set( $bool   = $myList.add( 3 ) )
#set( $bool   = $myList.add( "Hello" ) )
```

- To access an item, use its index:

```
$myList[ 0 ]
## or
$myList.get( 0 )
```

- To remove an item, use the `remove` method:

```
#set( $bool = $myList.remove( "Hello" ) )
```

- `java.util.ArrayList<E>`
- list.vm

Lists Returned by `selectNodes`

- When invoked through the `$_XPathTool` object, `selectNodes` always returns a list, which can be an empty list
- The type of such a list, even when empty, is `java.util.ArrayList<E>`, which is subclass of `java.util.AbstractCollection<E>`

Lists Returned by `getChildren`

- When invoked through an `org.jdom.Element` object, `getChildren` always returns a list, which can be an empty list
- The type of such a list, even when empty, is `org.jdom.ContentList$FilterList`, which is subclass of `java.util.AbstractCollection`
- Note that all three types of lists (list literals, lists returned by `org.jdom.Element.getChildren` and lists returned by `$_XPathTool.selectNodes`) are of subtypes of `java.util.AbstractCollection<E>`
- `#foreach` is designed to work with `java.util.AbstractCollection<E>` objects

Map Literals

- A map can be created by using the map notation `{}`

```
#set( $months = { "January":"janvier", "February":"février" } )
```

- The type of such a map is `java.util.LinkedHashMap<K,V>`
- Every entry in the map is a key-value pair, separated by a comma; a pair is separated by a colon
- A key can be an object of any type; normally we use Strings
- A value can be an object of any type
- We use the key to retrieve its corresponding value
- To work with a map, using loops, we normally need to retrieve the key set of the map:

```
#set( $months = { "January":"janvier", "February":"février" } )
#set( $engMonths = $months.keySet() )
```

The type of the key set is `java.util.LinkedHashMap$LinkedKeySet<K>`, which is subclass of `java.util.AbstractCollection<E>`.

- Note that even though the three types of lists are instances of three different classes, these three classes are descendants of `java.util.AbstractCollection<E>` and the `#foreach` structure works with all of them
- If we don't care about the keys, we can retrieve the values directly by calling `values`:

```
#set( $number = {
    1 : "one",
    2 : "two",
    3 : "three"
} )

#foreach( $value in $number.values() )
$value
#end
```

- **Important:**
  - An `java.util.LinkedHashMap$LinkedKeySet<K>` object maintains the order of the keys (the order of insertion)
  - That is to say, how we add key-value pairs to a map can matter if we want to rely on the order
  - Identity of lists and maps depends only on membership: two lists are considered identical if they contains exactly the same set of members, even though the members are ordered differently:

```
#set( $number1 = {
    1 : "one",
    2 : "two",
    3 : "three"
} )

#set( $number2 = {
    2 : "two",
    1 : "one",
    3 : "three"
} )
$number1.keySet() ## [1, 2, 3]
$number2.keySet() ## [2, 1, 3]

#if( $number1.keySet() == $number2.keySet() ) ## true
The two key sets are equal.
#end

#if( $number1 == $number2 ) ## true
The two maps are equal
#end
```

- java.util.LinkedHashMap<K,V>
- map.vm

Loops

- The `java.util.AbstractCollection<E>` class has an `iterator` method that works with the Velocity `#foreach` directive
- The `#foreach` structure:

```
#foreach( $item in $list )

#end
```

- For the `#foreach` structure to work, the type of the `$list` variable must be `java.util.AbstractCollection<E>`
- That is to say, the type of the list can be one of the following:
  - `java.util.ArrayList<E>`
  - `org.jdom.ContentList$FilterList`
  - `java.util.LinkedHashMap$LinkedKeySet<K>`
- Before we start working with the `#foreach` directive, we can use `$list.Class.Name` to

reveal class information

- Note that if the variable `$list` is undefined, or if it is of the wrong type, no error message will be issued and the Velocity engine will simply skip the `#foreach` directive; therefore, always make sure that we are dealing with a non-empty list before debugging a loop
- When the list is not empty, the local variable `$item` will store one of the items in the list, and there is a different current items in every loop
- As long as we are sure that an expression is evaluated to a non-empty list, the expression can appear right after `in`:

```
#foreach( $page in $contentRoot.getChildren( "system-page" ) )
    ## process page
#end

#set( $months = { "January":"janvier", "February":"février" } )

#foreach( $engMonth in $months.keySet() )
    $months[ $engMonth ]
#end
```

- A `#foreach` structure can be nested in another `#foreach` structure
- Besides the local variable declared in the `#foreach` directive, there is another variable supplied by the Velocity engine, namely, `$foreach`, whose type is `org.apache.velocity.runtime.directive.ForeachScope`
- Each `#foreach` structure has its own separate and distinct `$foreach` variable
- Important methods of `org.apache.velocity.runtime.directive.ForeachScope`, a subclass of `org.apache.velocity.runtime.directive.Scope`:
  - `int getCount()`: returns the 1-based count of the current item in the list
  - `boolean getFirst()`: returns a boolean, indicating whether the current loop is the first loop
  - `boolean getHasNext()`: same as `boolean hasNext()`
  - `int getIndex()`: returns the 0-based index of the current item in the list
  - `boolean getLast()`: returns a boolean, indicating whether the current loop is the last loop
  - `boolean hasNext()`: returns a boolean, indicating whether there is a loop following the current one; the value should be the negation of `getLast()`
  - `boolean isFirst()`: same as `boolean getFirst()`
  - `boolean isLast()`: same as `boolean getLast()`
- Important methods inherited from the parent class `org.apache.velocity.runtime.directive.Scope`:
  - `org.apache.velocity.runtime.directive.Scope getParent()`: returns the parent

`$foreach` object (the local variable of the parent `#foreach` structure)

- `org.apache.velocity.runtime.directive.Scope getTopmost()` : returns the `$foreach` object of the topmost `#foreach` structure

Using `$foreach`

- The `$foreach` object can be used to retrieve loop-related information for displaying purposes as well as to control the behavior of the loops
- `$foreach.Count` gives the 1-based count of the current item:

```
#set( $states   = [ 'NY', 'NJ', 'WA'  ] )

#foreach( $state in $states )
$foreach.Count $state
#end
```

This snippet outputs the following:

```
1 NY
2 NJ
3 WA
```

- `$foreach.Index` gives the 0-based count of the current item:

```
#set( $states   = [ 'NY', 'NJ', 'WA'  ] )
#set( $capitals = [ 'Albany', 'Trenton', 'Olympia' ] )

#foreach( $state in $states )
$foreach.Count $state $capitals[ $foreach.Index ]
#end
```

Here the `$foreach.Index` value is used to retrieve the corresponding item in another list. This snippet outputs the following:

```
1 NY Albany
2 NJ Trenton
3 WA Olympia
```

- We can check a property of the `$foreach` object to break out of a loop, using the `#break` directive:

```
#set( $list = [ 1..5 ] )

#foreach( $itemTop in $list )
$foreach.Count

    #if( $foreach.Count + 2 >= $list.size() )
        #break
    #end
#end
```

Here we want to skip the last two items. This snippet outputs the following:

```
1
2
3
```

- `$foreach.hasNext()` :
  - Can be used to peek ahead and to do something extra
  - Example: output a message before the looping is done

    ```
    <pre>
    #foreach( $num in [ 1..5 ] )
    Index: $foreach.Index
    #if( !$foreach.hasNext )We are done here!#end
    #end
    </pre>
    ```

    And the result:

    ```
    Index: 0
    Index: 1
    Index: 2
    Index: 3
    Index: 4
    We are done here!
    ```

- Since `#foreach` structures can be embedded, to access the properties of other layers of the `#foreach` structure, we can use `$foreach.Topmost` and `$foreach.Parent` :

```
#foreach( $itemTop in [ 1..3 ] )
    #foreach( $itemMiddle in [ 1..2 ] )
        #foreach( $itemInner in [ 1..3 ] )
            $foreach.Topmost.Count, $foreach.Parent.Count, $foreach.Count
        #end
    #end
#end
```

This snippet outputs the following:

```
1, 1, 1
1, 1, 2
1, 1, 3
1, 2, 1
1, 2, 2
1, 2, 3
2, 1, 1
2, 1, 2
2, 1, 3
2, 2, 1
2, 2, 2
2, 2, 3
3, 1, 1
3, 1, 2
3, 1, 3
3, 2, 1
3, 2, 2
3, 2, 3
```

- We can also use one of these `$foreach` objects to break out of any layer:

```
#foreach( $itemTop in [ 1..3 ] )
    #foreach( $itemMiddle in [ 1..2 ] )
        #foreach( $itemInner in [ 1..3 ] )
            #if( $foreach.Topmost.Count == 3 )
                #break( $foreach.Topmost )
            #end
            $foreach.Topmost.Count, $foreach.Parent.Count, $foreach.Count
        #end
    #end
#end
```

This snippet will break out of the topmost layer when the topmost count reaches 3, and it outputs the following:

```
1, 1, 1
1, 1, 2
1, 1, 3
1, 2, 1
1, 2, 2
1, 2, 3
2, 1, 1
2, 1, 2
2, 1, 3
2, 2, 1
2, 2, 2
2, 2, 3
```

Note that the `#break` directive is used here with `$foreach.Topmost`.

- Using `$foreach.Parent.hasNext`: we want to output different messages depending on

whether the parent has more items to process:

```
<pre>
#foreach( $num_outer in [ 1..3 ] )
Outer count: $foreach.Count
#foreach( $num_inner in [ 1..3 ] )
  Inner count: $foreach.Count
#if( !$foreach.hasNext && $foreach.parent.hasNext )  More to come!$n#elseif( !$forea
#end
#end
</pre>
```

This snippet outputs the following:

```
Outer count: 1
  Inner count: 1
  Inner count: 2
  Inner count: 3
  More to come!
Outer count: 2
  Inner count: 1
  Inner count: 2
  Inner count: 3
  More to come!
Outer count: 3
  Inner count: 1
  Inner count: 2
  Inner count: 3
  Finishing up!
```

- We can also use the `#stop` directive to stop the execution altogether:

```
<pre>
#foreach( $num_outer in [ 1..3 ] )
Outer count: $foreach.Count
#foreach( $num_inner in [ 1..3 ] )
  Inner count: $foreach.Count
#if( !$foreach.hasNext )${n}Sorry to interrupt!</pre>#stop #end
#end
#end
```

And the result:

```
Outer count: 1
  Inner count: 1
  Inner count: 2
  Inner count: 3

Sorry to interrupt!
```

Questions: Why does the end tag of `pre` appear right before `#stop`, and why are there curly brackets around `n` in `${n}`?

- Note that while `#break` causes the execution of the code to break out of a certain loop, `#stop` terminates the execution of the code altogether

---

Working with Lists

- The base class of all list instances in Velocity is `java.util.AbstractCollection<E>`
- Important methods of `java.util.AbstractCollection<E>`:
  - `boolean add(E e)`
  - `boolean addAll(Collection<? extends E> c)`
  - `void clear()`
  - `boolean contains(Object o)`
  - `boolean isEmpty()`
  - `boolean remove(Object o)`
  - `int size()`
- Because many methods return a boolean value, to avoid the returned value to be output, assign the returned value to a dummy variable:

```
#set( $void = $list.add( "this" ) )
```

- Note that this abstract class does not define a `get` method; the only way to work with such an object is to use `#foreach` to loop through all items
- The subclass `java.util.ArrayList<E>` do have a `get( int index )` method

Working with Maps

- The class of a map is `java.util.LinkedHashMap<K,V>`
- The parent class of `java.util.LinkedHashMap<K,V>` is `java.util.HashMap<K,V>`
- Important methods of these two classes:
  - `void clear()`
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object value)`
  - `V get(Object key)`
  - `boolean isEmpty()`

- `Set<K> keySet()`
- `V put(K key, V value)`
- `V putIfAbsent(K key, V value)`
- `boolean remove(Object key)`
- `V replace(K key, V value)`
- `boolean replace(K key, V oldValue, V newValue)`
- `int size()`
- `Collection<V> values()`

- When we need to access the keys of a map, use the method `keySet` :

```
#set( $keys = $map.keySet() )

#foreach( $key in $keys )
    ## work with both $key and $map[ $key ]
#end
```

- For example, we may want to sort the keys before accessing the values
- If we don't care about the keys and simply want to work with the values in any order, then we can use `#foreach` to loop through all values:

```
#foreach( $value in $map )
    ## work with the value
#end
```

Reinitializing Variables in Loops

- Recall that when the RHS of an assignment is evaluated to `null` , the assignment will fail
- When the assignment fails, the LHS keeps its old value, whatever it is (it could be `null`)
- If there are `#set` directives within a loop, it is very important to reinitialize all variables before using them, because they can have values preserved from a previous loop
- It is impossible to assign `null` to a variable
- A good alternative is to assign the empty String to all variables, and test for the empty String later
- Example:

```
#macro( processAsideGroup $asideGroup )
    #set( $asideGroupBlocks = $asideGroup.getChildren( "aside-group-chooser" ) )

    #foreach( $asideGroupBlock in $asideGroupBlocks )
        ## before using the variable, assign the empty String
        #set( $asideGroupBlockContent = '' )
        ## then try the assignment
        #set( $asideGroupBlockContent = $asideGroupBlock.getChild( 'content' ) )

        ## test for empty String; if not, proceed
        #if( $asideGroupBlockContent != '' )
            #processBlockChooser( $asideGroupBlockContent )
        #end
    #end
#end
```

- If assignments are done within conditionals, then try to introduce `#else` to pick up the default case
- Reinitialization of variables may also be required inside a macro if the macro is invoked in a `#foreach` structure and uses `#set` to assign values to global variables

---

Implementing a While Loop

- There is no `while` structure in Velocity
- It is still possible to implement a structure that has the effect of a `while` loop
- Basic setup:
  - Create a stopping integer value; e.g. `#set( $maxInt = 50000 )`
  - Use `#foreach( $num in [ 1..$maxInt ] )` to start looping
  - Within the `#foreach` structure, introduce an `#if…#else` structure
  - Put in a stopping condition (when met, stop looping) in the `#if` part, and use `#break` to exit
  - Put the code you want to be executed in loops in the `#else` part
- Example:

```
#set( $intMax = 50000 )

#foreach( $num in [ 1..$intMax ] )
    #if( $num == 10 )
        #break
    #else
        $num
    #end
#end
```

Lists vs. Arrays

- Do not mix up lists (`java.util.ArrayList` objects) and arrays
- A list is associated with a set of methods, whereas an array does not
- It does not make any sense to invoke methods through an array
- The type information is different
- For example, the structured data associated with a page is in fact an array

```
#set( $sd = $_.locatePage( "index", "cascade-admin" ).StructuredData )
$sd.Class.Name ##=> [Lcom.hannonhill.cascade.api.asset.common.StructuredDataNode;
```

- Note the `[L` and `;` in the type information

Important Points

- When working with a conditional, always provide an `#else` , even if its existence is just for debugging purposes
- When working with a list, always check its size
- Make sure that a list is indeed a list, not an array
- `#foreach` works with both lists (empty or otherwise) and arrays
- The order of insertion of key-value pairs into a map may matter
- Identity of lists and maps is determined by membership only
- Reinitialization of variables in loops is critical

Examples

- introductory/04_conditionals_and_loops

References

- Image else/if statment