EnterWorks Advanced VTL Mapping Guide

Revised 10/2/2020

Winshuttle

19820 North Creek Pkwy #200
Bothell, WA 98011

© Winshuttle
19820 North Creek Pkwy #200
Bothell, WA 98011

1.888.242.8356 (Sales and General Information)
1.888.225.2705 (U.S. Support)
http://www.enterworks.com

EnterWorks® Classic Administration Guide

Table of Contents

1 Document Conventions

This EnterWorks document uses the following typographic conventions:

Convention	Usage
pathnames	Pathnames are shown with backslashes, as for Windows systems.
`Courier New font`	Denotes sample code, for example, Java, IDL, and command line information. May be used to denote filenames and pathnames, calculations, code samples, registry keys, URLs, messages displayed on the screen. If *italicized* and in angle brackets (< >), it denotes a variable.
Calibri Font (bold)	When used in body text, it denotes an object, area, list item, button, or menu option within the graphical user interface; or a database name or database-related object. (Examples: the **Save** button; the **Product** tab; the **Name** field; the **SKU** repository) Can also be used to denote text that is typed in a text box. (Example: Type "**trackingNo**" in the Name field)
Blue underlined text	Words, phrases or numbers in blue are active links that can be clicked. Clicking these active links will bring the user to the required information, steps, pages chapters, or URL.

2 Document Terminology

This document uses the following terminology:

Convention	Usage
“EnterWorks” and “Enable”	The EnterWorks Enable product is now referred to simply as “EnterWorks”. Some system components and images in this document may still retain the name “Enable”.

3 Customer Support

EnterWorks provides a full spectrum of customer support. Check your maintenance contract for details about the level of support purchased. A customer identification number will be issued the first time customer support is contacted. Keep this number for future reference when using the EnterWorks customer support service.

How to reach us	Comments
On the Web: https://enterworkssupport.zendesk.com	Our knowledge base includes solutions to common issues and is available 24 hours a day, 7 days a week.
Create a Support Request Ticket: https://enterworkssupport.zendesk.com/hc/en-us/requests/new	You can generate a Support Request ticket at any time. The Winshuttle Support team addresses Support Request tickets during our normal business hours: • 5 AM – 5 PM Pacific Time • Monday-Friday
Postal mail: Winshuttle Customer Support Team 19820 North Creek Pkwy #200 Bothell, WA 98011 USA	Please include your preferred contact information, as well as a description of your request.

4 About this Guide

This guide is for users who are creating and editing advanced Enterworks layout mapping definitions using the open source Velocity Template Language (VTL). Velocity can be used to generate Web pages, SQL, PostScript and other output forms. EnterWorks incorporates the Velocity engine for the complex transformation of source content (such as product information) to the form used in target publications.

5 Velocity Template Language (VTL): An Introduction

The Velocity Template Language (VTL) is meant to provide the easiest, simplest, and cleanest way to incorporate dynamic content for the mapping of publication content. Even a user with little or no programming experience should be able to use VTL to incorporate dynamic content in a publication layout mapping.

EnterWorks uses VTL to *reference*, concatenate, and embed dynamic content (PIM attributes) along with scripting logic to control output mappings. Here is an example of a VTL statement that could be embedded in a mapping statement:

```
#set ($a = "Velocity”)
```

Like all VTL statements, this statement begins with the *#* character and contains a directive: *set*. When an online visitor requests your Web page, the Velocity Templating Engine will search through your Web page to find all *#* characters, then determine which mark the beginning of VTL statements, and which of the *#* characters that have nothing to do with VTL.

The *#* character is followed by a directive, *set*. The *set* directive uses an expression (enclosed in brackets) -- an equation that assigns a *value* to a *variable*. The variable is listed on the left hand side, and its value is listed on the right hand side; the two are separated by an = character.

In the example above, the variable is *$a* and the value is *Velocity*. This variable, like all references, begins with the *$* character. String values are always enclosed in quotes, either single or double quotes. Single quotes will ensure that the quoted value will be assigned to the reference as is. Double quotes allow you to use Velocity references and directives to interpolate, such as "Hello $name", where the *$name* will be replaced by the current value before that string literal is assigned to the left hand side of the =

The following rule of thumb may be useful to better understand how Velocity works:

References begin with $ and are used to <u>get</u> something. Directives begin with # and are used to <u>do</u> something.

In the example above, *#set* is used to assign a value to a variable. The variable *$a* can then be used in the template to output “Velocity.”

5.1 Comments

Comments allows descriptive text to be included that is not placed into the output of the template engine. Comments are a useful way of reminding yourself and explaining to others what your VTL statements are doing, or any other purpose you find useful. Below is an example of a comment in VTL.

```
## This is a single line comment.
```

A single line comment begins with *##* and finishes at the end of the line. If you're going to write a few lines of commentary, there's no need to have numerous single line comments. Multi-line comments, which begin with *#** and end with ***#**, are available to handle this scenario.

```
This is text that is outside the multi-line comment.
Online visitors can see it.

#*
 This begins a multi-line comment. Online visitors won't
 see this text because the Velocity Templating Engine will
 ignore it.
*#

Here is text outside the multi-line comment; it is visible.
```

Here are a few examples to clarify how single line and multi-line comments work:

```
This text is visible. ## This text is not.
This text is visible.
This text is visible. #* This text, as part of a multi-line
comment, is not visible. This text is not visible; it is also
part of the multi-line comment. This text still not
visible. *# This text is outside the comment, so it is visible.
## This text is not visible.
```

There is a third type of comment, the VTL comment block, which may be used to store any sort of extra information you want to track in the template (such as Javadoc-style author and versioning information):

```
#**
This is a VTL comment block and
may be used to store such information
```

```
as the document author and versioning
information:
@author
@version 5
*#
```

5.2 References

There are three types of references in VTL: variables, properties, and methods. As a designer using VTL, you and your developers must come to an agreement on the specific names of references so you can use them correctly in your templates.

5.3 Variables

The shorthand notation of a variable consists of a leading "$" character followed by a VTL *Identifier*. A VTL Identifier must start with an alphabetic character (a ... z or A ... Z). The rest of the characters are limited to the following types of characters:

- alphabetic (a .. z, A .. Z)
- numeric (0 .. 9)
- hyphen ("-")
- underscore ("_")

Here are some examples of valid variable references in the VTL:

```
$foo
$mudSlinger
$mud-slinger
$mud_slinger
$mudSlinger1
```

When VTL references a variable, such as *$foo*, the variable can get its value from either a *set* directive in the template, or from the Java code. For example, if the Java variable *$foo* has the value *bar* at the time the template is requested, *bar* replaces all instances of *$foo* on the Web page. Alternatively, if you include the statement

```
#set ($foo = "bar")
```

The output will be the same for all instances of $foo that follow this directive.

5.4 Properties

The second category of VTL references is properties, which have a distinctive format. The shorthand notation consists of a leading *$* character followed a VTL Identifier, followed by a dot character (“.”) and another VTL Identifier. These are examples of valid property references in the VTL:

```
$customer.Address
$purchase.Total
```

Take the first example, *$customer.Address*. It can have two meanings. It can mean look in the hashtable identified as *customer* and return the value associated with the key *Address*. But *$customer.Address* can also be referring to a method (references that refer to methods will be discussed in the next section); *$customer.Address* could be an abbreviated way of writing *$customer.getAddress()*. When your page is requested, Velocity will determine which of these two possibilities makes sense, and then return the appropriate value.

5.5 Methods

Methods are references that consist of a leading “$” character followed a VTL Identifier, followed by a VTL *Method Body*. A VTL Method Body consists of a VTL Identifier followed by a left parenthesis character (“(“), followed by an optional parameter list, followed by right parenthesis character (“)”). These are examples of valid method references in VTL:

```
$customer.getAddress()
$purchase.getTotal()
$page.setTitle( "My Home Page" )
$person.setAttributes( ["Strange", "Weird", "Excited"] )
```

The first two examples -- *$customer.getAddress()* and *$purchase.getTotal()* -- may look similar to those used in the Properties section above, *$customer.Address* and *$purchase.Total*. If you guessed that these examples must be related some in some fashion, you’re correct!

VTL Properties can be used as a shorthand notation for VTL Methods. The Property *$customer.Address* has the exact same effect as using the Method *$customer.getAddress()*. It is generally preferable to use a Property when available. The main difference between Properties and Methods is that you can specify a parameter list to a Method.

The shorthand notation can be used for the following Methods:

```
$sun.getPlanets()
$annelid.getDirt()
$album.getPhoto()
```

We might expect these methods to return the names of planets belonging to the sun, feed our earthworm, or get a photograph from an album. Only the long notation works for the following Methods.

```
$sun.getPlanet( ["Earth", "Mars", "Neptune"] )
## Can't pass a parameter list with $sun.Planets

$sisyphus.pushRock()
## Velocity assumes I mean $sisyphus.getRock()

$book.setTitle( "Homage to Catalonia" )
## Can't pass a parameter
```

As of Velocity 1.6, all array references are now “magically” treated as if they are fixed-length lists. This means that you can call java.util.List methods on array references. So, if you have a reference to an array (let’s say this one is a String[] with three values), you can specify:

```
$myarray.isEmpty()
$myarray.size()
$myarray.get(2)
$myarray.set(1, 'test')
```

Also new in Velocity 1.6 is support for vararg methods. A method like:

```
public void setPlanets(String... planets)
```

or even just:

```
public void setPlanets(String[] planets)
```

(if you are using a pre-Java 5 JDK), can now accept any number of arguments when called in a template:

```
$sun.setPlanets('Earth', 'Mars', 'Neptune')
$sun.setPlanets('Mercury')
$sun.setPlanets()
```

```
## Will just pass in an empty, zero-length array
```

Property Lookup Rules

As was mentioned earlier, properties often refer to methods of the parent object. Velocity is quite clever when figuring out which method corresponds to a requested property. It tries out different alternatives based on several established naming conventions. The exact lookup sequence depends on whether the property name starts with an upper-case letter. For lower-case names, such as *$customer.address*, the sequence is

1. getaddress()
2. getAddress()
3. get("address")
4. isAddress()

For upper-case property names like *$customer.Address*, it's slightly different:

1. getAddress()
2. getaddress()
3. get("Address")
4. isAddress()

Rendering

The final value resulting from each and every reference (whether variable, property, or method) is converted to a String object when it is rendered into the final output. If there is an object that represents *$foo* (such as an Integer object), then Velocity will call its `.toString()` method to resolve the object into a String.

Formal Reference Notation

Shorthand notation for references is used in the examples above, but there also is a formal notation for references:

```
${mudSlinger}
${customer.Address}
${purchase.getTotal()}
```

In almost all cases, you'll use the shorthand notation for references, but in some cases the formal notation is required for correct processing.

Suppose you're constructing a sentence on the fly where *$vice* is used as the base word in the noun of a sentence. The goal is to allow someone to choose the base word and produce one of the two following results: "Jack is a pyromaniac." or "Jack is a kleptomaniac." Using the shorthand notation would be inadequate for this task. Consider the following example:

```
Jack is a $vicemaniac.
```

There is ambiguity here, and Velocity assumes that *$vicemaniac*, not *$vice*, is the Identifier that you mean to use. Finding no value for *$vicemaniac*, it will return *$vicemaniac*. Using formal notation can resolve this problem.

```
Jack is a ${vice}maniac.
```

Now Velocity knows that *$vice*, not *$vicemaniac*, is the reference. Formal notation is often useful when references are directly adjacent to text in a template.

Quiet Reference Notation

When Velocity encounters an undefined reference, its normal behavior is to output the image of the reference. For example, suppose the following reference appears as part of a VTL template.

```
<input type="text" name="email" value="$email"/>
```

When the form initially loads, the variable reference *$email* has no value, but you prefer a blank text field to one with a value of "$email". Using the quiet reference notation circumvents Velocity's normal behavior; instead of using *$email* in the VTL you would use *$!email*. So the above example would look like the following:

```
<input type="text" name="email" value="$!email"/>
```

Now when the form is initially loaded and *$email* still has no value, an empty string will be output instead of "$email".

Formal and quiet reference notation can be used together, as demonstrated below.

```
<input type="text" name="email" value="$!{email}"/>
```

5.6 Strict References Setting

Velocity 1.6 introduces the concept of strict references, which is activated by setting the velocity configuration property 'runtime.references.strict' to true. With this setting references are required to be either placed explicitly into the context or defined with a #set directive, or Velocity will throw an exception. References that are in the context with a value of null will not produce an exception. Additionally, if an attempt is made to call a method or a property on an object within a reference that does not define the specified method or property then Velocity will throw an exception. This is also true if there is an attempt to call a method or property on a null value.

In the following examples $bar is defined, but $foo is not, and all these statements will throw an exception:

```
$foo                            ## Exception
#set($bar = $foo)               ## Exception
#if($foo == $bar)#end           ## Exception
#foreach($item in $foo)#end     ## Exception
```

Also, the following statements show examples in which Velocity will throw an exception when attempting to call methods or properties that do not exist. In these examples $bar contains an object that defines a property 'foo' which returns a string, and 'retnull' which returns null.

```
$bar.bogus          ## $bar does not provide property bogus, Exception
$bar.foo.bogus      ## $bar.foo does not provide property bogus,
Exception
$bar.retnull.bogus  ## cannot call a property on null, Exception
```

In general, strict reference behavior is true for all situations in which references are used, except for a special case within the #if directive. If a reference is used within a #if or #elseif directive without any methods or properties, and if it is not being compared to another value, then undefined references are allowed. This behavior provides an easy way to test if a reference is defined before using it in a template. In the following example, where $foo is not defined, the statements will not throw an exception.

```
#if ($foo)#end                  ## False
#if ( ! $foo)#end               ## True
#if ($foo && $foo.bar)#end      ## False and $foo.bar will not be
evaluated
#if ($foo && $foo == "bar")#end ## False and $foo == "bar" will not be
evaluated
```

```
#if ($foo1 || $foo2)#end        ## False $foo1 and $foo2 are not
defined
```

One additional note: undefined macro references will also throw an exception with the strict reference setting.

5.7 Case Substitution

Now that you are familiar with references, you can begin to apply them effectively in your templates. Velocity references take advantage of some Java principles that template designers will find easy to use. For example:

```
$foo

$foo.getBar()
## is the same as
$foo.Bar

$data.setUser("jon")
## is the same as
#set( $data.User = "jon" )

$data.getRequest().getServerName()
## is the same as
$data.Request.ServerName
## is the same as
${data.Request.ServerName}
```

These examples illustrate alternative uses for the same references. Velocity takes advantage of Java's introspection and Bean features to resolve the reference names to both objects in the Context as well as the objects methods. It is possible to embed and evaluate references almost anywhere in your template.

Velocity, which is modeled on the Bean specifications defined by Sun Microsystems, is case sensitive; however, its developers have strived to catch and correct user errors wherever possible. When the method *getFoo()* is referred to in a template by *$bar.foo*, Velocity will first try *$getfoo*. If this fails, it will then try *$getFoo*. Similarly, when a template refers to *$bar.Foo*, Velocity will try *$getFoo()* first and then try *getfoo()*.

Note: *References to instance variables in a template are not resolved.* Only references to the attribute equivalents of JavaBean getter/setter methods are resolved (i.e. `$foo.Name` does resolve to the class Foo's *getName()* instance method, but not to a public *Name* instance variable of Foo).

5.8 Directives

References allow template designers to generate dynamic content for Web sites, while *directives* -- easy to use script elements that can be used to creatively manipulate the output of Java code -- permit Web designers to truly take charge of the appearance and content of the Web site.

Directives always begin with a `#`. Like references, the name of the directive may be bracketed by "`{`" and "`}`" symbols. This is useful with directives that are immediately followed by text. For example the following produces an error:

```
#if($a==1)true enough#elseno way!#end
```

In such a case, use the brackets to separate `#else` from the rest of the line.

```
#if($a==1)true enough#{else}no way!#end
```

#set

The *#set* directive is used for setting the value of a reference. A value can be assigned to either a variable reference or a property reference, and this occurs in parentheses:

```
#set( $primate = "monkey" )
#set( $customer.Behavior = $primate )
```

The left hand side (LHS) of the assignment must be a variable reference or a property reference. The right hand side (RHS) can be one of the following types:

- Variable reference
- String literal
- Property reference
- Method reference
- Number literal
- ArrayList
- Map

These examples demonstrate each of the aforementioned types:

```
#set( $monkey = $bill ) ## variable reference
```

```
#set( $monkey.Friend = "monica" ) ## string literal
#set( $monkey.Blame = $whitehouse.Leak ) ## property reference
#set( $monkey.Plan = $spindoctor.weave($web) ) ## method
reference
#set( $monkey.Number = 123 ) ##number literal
#set( $monkey.Say = ["Not", $my, "fault"] ) ## ArrayList
#set( $monkey.Map = {"banana" : "good", "roast beef" : "bad"})
## Map
```

NOTE: For the ArrayList example the elements defined with the [..] operator are accessible using the methods defined in the ArrayList class. So, for example, you could access the first element above using $monkey.Say.get(0).

Similarly, for the Map example, the elements defined within the { } operator are accessible using the methods defined in the Map class. So, for example, you could access the first element above using $monkey.Map.get("banana") to return a String 'good', or even $monkey.Map.banana to return the same value.

The RHS can also be a simple arithmetic expression:

```
#set( $value = $foo + 1 )
#set( $value = $bar - 1 )
#set( $value = $foo * $bar )
#set( $value = $foo / $bar )
```

If the RHS is a property or method reference that evaluates to *null*, it will **not** be assigned to the LHS. Depending on how Velocity is configured, it is usually not possible to remove an existing reference from the context via this mechanism. (Note that this can be done by changing one of the Velocity configuration properties). This can be confusing for newcomers to Velocity. For example:

```
#set( $result = $query.criteria("name") )
The result of the first query is $result

#set( $result = $query.criteria("address") )
The result of the second query is $result
```

If *$query.criteria("name")* returns the string "bill", and *$query.criteria("address")* returns *null*, the above VTL will render as the following:

```
The result of the first query is bill

The result of the second query is bill
```

This tends to confuse newcomers who construct *#foreach* loops that attempt to *#set* a reference via a property or method reference, then immediately test that reference with an *#if* directive. For example:

```
#set( $criteria = ["name", "address"] )
#foreach( $criterion in $criteria )
    #set( $result = $query.criteria($criterion) )
    #if( $result )
        Query was successful
    #end
#end
```

In the above example, it would not be wise to rely on the evaluation of *$result* to determine if a query was successful. After *$result* has been *#set* (added to the context), it cannot be set back to *null* (removed from the context). The details of the *#if* and *#foreach* directives are covered later in this document.

One solution is to pre-set *$result* to *false*. Then if the *$query.criteria()* call fails, you can check.

```
#set( $criteria = ["name", "address"] )

#foreach( $criterion in $criteria )

    #set( $result = false )
    #set( $result = $query.criteria($criterion) )

    #if( $result )
        Query was successful
    #end

#end
```

Unlike some of the other Velocity directives, the *#set* directive does not have an *#end* statement.

Literals

When using the *#set* directive, string literals that are enclosed in double quote characters will be parsed and rendered, as shown:

```
#set( $directoryRoot = "www" )
```

```
#set( $templateName = "index.vm" )
#set( $template = "$directoryRoot/$templateName" )
$template
```

The output will be:

```
www/index.vm
```

However, when the string literal is enclosed in single quote characters, it will not be parsed:

```
#set( $foo = "bar" )
$foo
#set( $blargh = '$foo' )
$blargh
```

This renders as:

```
bar
$foo
```

By default, this feature of using single quotes to render unparsed text is available in Velocity. This default can be changed by editing `velocity.properties` such that `stringliterals.interpolate=false`.

Alternatively, the *#literal* script element allows the template designer to easily use large chunks of uninterpreted content in VTL code. This can be especially useful in place of escaping multiple directives:

```
#literal()
#foreach ($woogie in $boogie)
  nothing will happen to $woogie
#end
#end
```

Renders as:

```
#foreach ($woogie in $boogie)
  nothing will happen to $woogie
#end
```

5.9 Conditionals

If / ElseIf / Else

The *#if* directive in Velocity allows for text to be included when the web page is generated, on the conditional that the if statement is true. For example:

```
#if( $foo )
   <strong>Velocity!</strong>
#end
```

The variable *$foo* is evaluated to determine whether it is true, which will happen under one of two circumstances: (i) *$foo* is a boolean (true/false) which has a true value, or (ii) the value is not null. Remember that the Velocity context only contains Objects, so when we say "boolean," it will be represented as a Boolean (the class). This is true even for methods that return `boolean` - the introspection infrastructure will return a `Boolean` of the same logical value.

The content between the *#if* and the *#end* statements becomes the output if the evaluation is true. In this case, if *$foo* is true, the output will be: "Velocity!" Conversely, if *$foo* has a null value, or if it is a boolean false, the statement evaluates as false, and there is no output.

An *#elseif* or *#else* element can be used with an *#if* element. Note that the Velocity Templating Engine will stop at the first expression that is found to be true. In the following example, suppose that *$foo* has a value of 15 and *$bar* has a value of 6.

```
#if( $foo < 10 )
    <strong>Go North</strong>
#elseif( $foo == 10 )
    <strong>Go East</strong>
#elseif( $bar == 6 )
    <strong>Go South</strong>
#else
    <strong>Go West</strong>
#end
```

In this example, *$foo* is greater than 10, so the first two comparisons fail. Next *$bar* is compared to 6, which is true, so the output is **Go South**.

Relational and Logical Operators

Velocity uses the equivalent operator to determine the relationships between variables. Here is a simple example to illustrate how the equivalent operator is used:

```
#set ($foo = "deoxyribonucleic acid")
#set ($bar = "ribonucleic acid")

#if ($foo == $bar)
  In this case it's clear they aren't equivalent. So...
#else
  They are not equivalent and this will be the output.
#end
```

Note that the semantics of == are slightly different than Java where == can only be used to test object equality. In Velocity the equivalent operator can be used to directly compare numbers, strings, or objects. When the objects are of different classes, the string representations are obtained by calling `toString()` for each object and then compared.

Velocity has logical AND, OR and NOT operators as well. Below are examples demonstrating the use of the logical AND, OR and NOT operators.

```
## logical AND

#if( $foo && $bar )
   <strong> This AND that</strong>
#end
```

The *#if()* directive will evaluate to true only if both *$foo* and *$bar* are true. If *$foo* is false, the expression will evaluate to false; *$bar* will not be evaluated. If *$foo* is true, the Velocity Templating Engine will then check the value of *$bar*; if *$bar* is true, then the entire expression is true and **This AND that** becomes the output. If *$bar* is false, then there will be no output as the entire expression is false.

Logical OR operators work the same way, except only one of the references needs to evaluate to true in order for the entire expression to be considered true. Consider the following example.

```
## logical OR

#if( $foo || $bar )
    <strong>This OR That</strong>
#end
```

If *$foo* is true, the Velocity Templating Engine has no need to look at *$bar*; whether *$bar* is true or false, the expression will be true, and **This OR That** will be output. If *$foo* is false, however, *$bar* must be checked. In this case, if *$bar* is also false, the expression evaluates to false and

there is no output. On the other hand, if *$bar* is true, then the entire expression is true, and the output is **This OR That**

With logical NOT operators, there is only one argument:

```
##logical NOT

#if( !$foo )
  <strong>NOT that</strong>
#end
```

Here, the if *$foo* is true, then *!$foo* evaluates to false, and there is no output. If *$foo* is false, then *!$foo* evaluates to true and **NOT that** will be output. Be careful not to confuse this with the *quiet reference $!foo* which is something altogether different.

There are text versions of all logical operators, including *eq*, *ne*, *and*, *or*, *not*, *gt*, *ge*, *lt*, and *le*.

One more useful note: when you wish to include text immediately following a *#else* directive you will need to use curly brackets immediately surrounding the directive to differentiate it from the following text. (Any directive can be delimited by curly brackets, although this is most useful for *#else*).

```
#if( $foo == $bar)it's true!#{else}it's not!#end</li>
```

5.10 Loops

Foreach Loop

The *#foreach* element allows for looping. For example:

```
<ul>
#foreach( $product in $allProducts )
    <li>$product</li>
#end
</ul>
```

This *#foreach* loop causes the *$allProducts* list (the object) to be looped over for all of the products (targets) in the list. Each time through the loop, the value from *$allProducts* is placed into the *$product* variable.

The contents of the *$allProducts* variable is a Vector, a Hashtable or an Array. The value assigned to the *$product* variable is a Java Object and can be referenced from a variable as such. For example, if *$product* was really a Product class in Java, its name could be retrieved by referencing the *$product.Name* method (*$Product.getName()*).

Lets say that *$allProducts* is a Hashtable. If you wanted to retrieve the key values for the Hashtable as well as the objects within the Hashtable, you can use code like this:

```
<ul>
#foreach( $key in $allProducts.keySet() )
    <li>Key: $key -> Value: $allProducts.get($key)</li>
#end
</ul>
```

Velocity provides an easy way to get the loop counter so that you can do something like the following:

```
<table>
#foreach( $customer in $customerList )
    <tr><td>$velocityCount</td><td>$customer.Name</td></tr>
#end
</table>
```

Velocity also now provides an easy way to tell if you are on the last iteration of a loop:

```
#foreach( $customer in $customerList )
    $customer.Name#if( $velocityHasNext ),#end
#end
```

The default name for the "has next" variable reference, which is specified in the velocity.properties file, is $velocityHasNext. The default name for the loop counter variable reference, which is specified in the velocity.properties file, is $velocityCount. By default the counter starts at 1, but this can be set to either 0 or 1 in the `velocity.properties` file. Here's what the loop counter properties section of the `velocity.properties` file appears:

```
# Default name of the loop counter
# variable reference.
directive.foreach.counter.name = velocityCount
directive.foreach.iterator.name = velocityHasNext

# Default starting value of the loop
# counter variable reference.
directive.foreach.counter.initial.value = 1
```

It's possible to set a maximum allowed number of times that a loop may be executed. By default there is no max (indicated by a value of 0 or less), but this can be set to an arbitrary number in the `velocity.properties` file. This is useful as a fail-safe.

```
# The maximum allowed number of loops.
directive.foreach.maxloops = -1
```

If you want to stop looping in a foreach from within your template, you can now use the #break directive to stop looping at any time:

```
## list first 5 customers only
#foreach( $customer in $customerList )
    #if( $velocityCount > 5 )
        #break
    #end
    $customer.Name
#end
```

5.11 Include

The *#include* script element allows the template designer to import a local file, which is then inserted into the location where the *#include* directive is defined. The contents of the file are not rendered through the template engine. For security reasons, the file to be included may only be under TEMPLATE_ROOT.

```
#include( "one.txt" )
```

The file to which the *#include* directive refers is enclosed in quotes. If more than one file will be included, they should be separated by commas.

```
#include( "one.gif","two.txt","three.htm" )
```

The file being included need not be referenced by name; in fact, it is often preferable to use a variable instead of a filename. This could be useful for targeting output according to criteria determined when the page request is submitted. Here is an example showing both a filename and a variable:

```
#include( "greetings.txt", $seasonalstock )
```

5.12 Parse

The *#parse* script element allows the template designer to import a local file that contains VTL. Velocity will parse the VTL and render the template specified.

```
#parse( "me.vm" )
```

Like the *#include* directive, *#parse* can take a variable rather than a template. Any templates to which *#parse* refers must be included under TEMPLATE_ROOT. Unlike the *#include* directive, *#parse* will only take a single argument.

VTL templates can have *#parse* statements referring to templates that in turn have *#parse* statements. By default set to 10, the *directive.parse.max.depth* line of the `velocity.properties` allows users to customize maximum number of *#parse* referrals that can occur from a single template. (Note: If the *directive.parse.max.depth* property is absent from the `velocity.properties` file, Velocity will set this default to 10.) Recursion is permitted, for example, if the template `dofoo.vm` contains the following lines:

```
Count down.
#set( $count = 8 )
#parse( "parsefoo.vm" )
All done with dofoo.vm!
```

It would reference the template `parsefoo.vm`, which might contain the following VTL:

```
$count
#set( $count = $count - 1 )
#if( $count > 0 )
    #parse( "parsefoo.vm" )
#else
    All done with parsefoo.vm!
#end
```

After “Count down” is displayed, Velocity passes through `parsefoo.vm`, counting down from 8. When the count reaches 0, it will display the “All done with parsefoo.vm!” message. At this point, Velocity will return to `dofoo.vm` and output the “All done with dofoo.vm!” message.

5.13 Stop

The *#stop* script element prevents any further text or references in the page from being rendered. This is useful for debugging purposes.

5.14 Evaluate

The *#evaluate* directive can be used to dynamically evaluate VTL. This allows the template to evaluate a string that is created at render time. Such a string might be used to internationalize the template or to include parts of a template from a database.

The example below will display `abc`.

```
#set($source1 = "abc")
#set($select = "1")
#set($dynamicsource = "$source$select")
## $dynamicsource is now the string '$source1'
#evaluate($dynamicsource)
```

5.15 Define

The *#define* directive lets one assign a block of VTL to a reference.

The example below will display `Hello World!`.

```
#define( $block )Hello $who#end
#set( $who = 'World!' )
$block
```

5.16 Getting Literal

VTL uses special characters, such as *$* and *#*, to do its work, so some added care should be taken where using these characters in your templates. This section deals with escaping these characters.

Currency
There is no problem writing “I bought a 4 lb. sack of potatoes at the farmer's market for only $2.50!” As mentioned, a VTL identifier always begins with an upper- or lower-case letter, so $2.50 would not be mistaken for a reference.

Escaping Valid VTL References
Cases may arise where you do not want to have a reference rendered by Velocity. *Escaping* special characters is the best way to output VTL's special characters in these situations, and this can be done using the backslash (\ ) character *when those special characters are part of a valid VTL reference* .

```
#set( $email = "foo" )
$email
```

If Velocity encounters a reference in your VTL template to *$email*, it will search the Context for a corresponding value. Here the output will be *foo*, because *$email* is defined. If *$email* is not defined, the output will be *$email*.

Suppose that *$email* is defined (for example, if it has the value *foo*), and that you want to output *$email*. There are a few ways of doing this, but the simplest is to use the escape character. Here is a demonstration:

```
## The following line defines $email in this template:
#set( $email = "foo" )
$email
\$email
```

This renders as:

```
foo
$email
```

If, for some reason, you need a backslash before either line above, you can do the following:

```
## The following line defines $email in this template:
#set( $email = "foo" )
\\$email
\\\$email
```

This renders as:

```
\foo
\$email
```

Note that the \ character bind to the *$* from the left. The bind-from-left rule causes *\\\$email* to render as *\$email*. Compare these examples to those in which *$email* is not defined.

```
$email
\$email
```

```
\\$email
\\\$email
```

renders as

```
$email
\$email
\\$email
\\\$email
```

Note that Velocity handles references that are defined differently from those that have not been defined. Here is a set directive that gives *$foo* the value *gibbous*.

```
#set( $foo = "gibbous" )
$moon = $foo
```

The output will be: *$moon = gibbous* -- where *$moon* is output as a literal because it is undefined and *gibbous* is output in place of *$foo*.

Escaping Invalid VTL References

Sometimes Velocity has trouble parsing your template when it encounters an "invalid reference" that you never intended to be a reference at all. *Escaping* special characters is, again, the best way to handle these situations. In these situations, the backslash will likely fail you. Instead of simply trying to escape the problematic `$` or `#`, you should probably just replace this:

```
${my:invalid:non:reference}
```

with something like this:

```
#set( $D = '$' )
${D}{my:invalid:non:reference}
```

You can, of course, put your `$` or `#` string directly into the context from your java code (e.g. `context.put("D","$");`) to avoid the extra #set() directive in your template(s). Or, if you are using VelocityTools , you can just use the EscapeTool like this:

```
${esc.d}{my:invalid:non:reference}
```

Escaping of both valid and invalid VTL directives is handled in much the same manner; this is described in more detail in the Directives section.

Escaping VTL Directives

VTL directives can be escaped with the backslash character ("\") in a manner similar to valid VTL references.

```
## #include( "a.txt" ) renders as <contents of a.txt>
#include( "a.txt" )

## \#include( "a.txt" ) renders as #include( "a.txt" )
\#include( "a.txt" )

## \\#include ( "a.txt" ) renders as \<contents of a.txt>
\\#include ( "a.txt" )
```

Extra care should be taken when escaping VTL directives that contain multiple script elements in a single directive (such as in an if-else-end statements). Here is a typical VTL if-statement:

```
#if( $jazz )
    Vyacheslav Ganelin
#end
```

If *$jazz* is true, the output is:

```
Vyacheslav Ganelin
```

If *$jazz* is false, there is no output. Escaping script elements alters the output. Consider the following case:

```
\#if( $jazz )
    Vyacheslav Ganelin
\#end
```

This causes the directives to be escaped, but the rendering of *$jazz* proceeds as normal. So, if *$jazz* is true, the output is:

```
#if( true )
    Vyacheslav Ganelin
#end
```

Suppose backslashes precede script elements that are legitimately escaped:

```
\\#if( $jazz )
   Vyacheslav Ganelin
Error! Hyperlink reference not valid.
```

In this case, if *$jazz* is true, the output is:

```
\ Vyacheslav Ganelin
\
```

To understand this, note that the `#if( arg )`, when ended by a newline (return), will omit the newline from the output. Therefore, the body of the `#if()` block follows the first "\", rendered from the "\\" preceding the `#if()`. The last \ is on a different line than the text because there is a newline after "Ganelin", so the final \\, preceding the `#end` is part of the body of the block.

If *$jazz* is false, the output is:

```
\
```

Note that things start to break if script elements are not properly escaped.

```
\\\#if( $jazz )
    Vyacheslave Ganelin
\\#end
```

Here the *#if* is escaped, but there is an *#end* remaining; having too many endings will cause a parsing error.

5.17 VTL: Formatting Issues

Although VTL in this user guide is often displayed with newlines and whitespaces, the VTL shown below:

```
#set( $imperial = ["Munetaka","Koreyasu","Hisakira","Morikune"]
)
#foreach( $shogun in $imperial )
    $shogun
#end
```

is equally valid as the following snippet that Geir Magnusson, Jr. posted to the Velocity user mailing list to illustrate a completely unrelated point:

```
Send me #set($foo=["$10 and ","a pie"])#foreach($a in
$foo)$a#end please.
```

Velocity's behavior is to gobble up excess whitespace. The preceding directive can be written as:

```
Send me
#set( $foo = ["$10 and ","a pie"] )
#foreach( $a in $foo )
$a
#end
please.
```

or as:

```
Send me
#set($foo       = ["$10 and ","a pie"])
                  #foreach           ($a in $foo )$a
         #end please.
```

In each case the output will be the same.

5.18 Other Features and Miscellany

5.18.1 Math

Velocity has a handful of built-in mathematical functions that can be used in templates with the *set* directive. The following equations are examples of addition, subtraction, multiplication and division, respectively:

```
#set( $foo = $bar + 3 )
#set( $foo = $bar - 4 )
#set( $foo = $bar * 6 )
#set( $foo = $bar / 2 )
```

When a division operation is performed between two integers, the result will be an integer, as the fractional portion is discarded. Any remainder can be obtained by using the modulus (*%*) operator:

```
#set( $foo = $bar % 5 )
```

5.18.2 Range Operator

The range operator can be used in conjunction with *#set* and *#foreach* statements. Useful for its ability to produce an object array containing integers, the range operator has the following construction:

```
[n..m]
```

Both *n* and *m* must either be or produce integers. Whether *m* is greater than or less than *n* will not matter; in this case the range will simply count down. Examples showing the use of the range operator as provided below:

```
First example:
#foreach( $foo in [1..5] )
$foo
#end

Second example:
#foreach( $bar in [2..-2] )
$bar
#end

Third example:
#set( $arr = [0..1] )
#foreach( $i in $arr )
$i
#end

Fourth example:
[1..3]
```

Produces the following output:

```
First example:
1 2 3 4 5

Second example:
2 1 0 -1 -2

Third example:
0 1

Fourth example:
[1..3]
```

Note that the range operator only produces the array when used in conjunction with *#set* and *#foreach* directives, as demonstrated in the fourth example.

Web page designers concerned with making tables a standard size, but where some will not have enough data to fill the table, will find the range operator particularly useful.

5.18.3 Advanced Issues: Escaping and !

When a reference is silenced with the *!* character and the *!* character preceded by an \ escape character, the reference is handled in a special way. Note the differences between regular escaping, and the special case where \ precedes *!* follows it:

```
#set( $foo = "bar" )
$\!foo
$\!{foo}
$\\!foo
$\\\!foo
```

This renders as:

```
$!foo
$!{foo}
$\!foo
$\\!foo
```

Contrast this with regular escaping, where \ precedes *$*:

```
\$foo
\$!foo
\$!{foo}
\\$!{foo}
```

This renders as:

```
$foo
$!foo
$!{foo}

\bar
```

6 Summarized VTL Guide

6.1 General

Variables

Notation:

$ [!][{][a..z, A..Z][a..z, A..Z, 0..9, -, _][}]

Examples:

- *Normal notation: $mud-Slinger_9*
- *Silent notation: $!mud-Slinger_9*
- *Formal notation: ${mud-Slinger_9}*

Properties

Notation:

$ [{][a..z, A..Z][a..z, A..Z, 0..9, -, _]* .[a..z, A..Z][a..z, A-Z, 0..9, -, _]* [}]

Examples:

- *Regular Notation: $customer.Address*
- *Formal Notation: ${purchase.Total}*

VTL Properties can be used as a shorthand notation for VTL Methods that take get and set. Either $object.getMethod() or $object.setMethod() can be abbreviated as $object.Method. It is generally preferable to use a Property when available. The main difference between Properties and Methods is that you can specify a parameter list to a Method.

Methods

Notation:

$** [**{**][**a..z**, **A..Z**][**a..z**, **A..Z**, **0..9**, **-**, **_**]* **.**[**a..z**, **A..Z**][**a..z**, **A..Z**, **0..9**, **-**, **_**](** [opional parameter list...] **)** [**}**]*

Examples:

- *Regular Notation: $customer.getAddress()*
- *Formal Notation: ${purchase.getTotal()}*
- *Regular Notation with Parameter List: $page.setTitle("My Home Page")*

6.2 Directives

#set - Establishes the value of a reference

Format:

#set($ref = ['', ']arg['', '])

Usage:

- *$ref* - The LHS of the assignment must be a variable reference or a property reference.
- *arg* - The RHS of the assignment, *arg* is parsed if enclosed in double quotes, and not parsed if enclosed in single quotes.

Examples:

- Variable reference: #set($monkey = "bill")
- String literal: #set($monkey.Friend = "monica")
- Property reference: #set($monkey.Blame = $whitehouse.Leak)
- Method reference: #set($monkey.Plan = $spindoctor.weave($web))
- Number literal: #set($monkey.Number = 123)
- Object array: #set($monkey.Say = ["Not", $my, "fault"])

The RHS can also be a simple arithmetic expression, such as:

- Addition: #set($value = $foo + 1)
- Subtraction: #set($value = $bar - 1)
- Multiplication: #set($value = $foo * $bar)
- Division: #set($value = $foo / $bar)
- Remainder: #set($value = $foo % $bar)

#if / #elseif / #else - output conditional on truth of statements

Format:

#if([condition] **)** [output] [**#elseif(** [condition] **)** [output]]* [**#else** [output]] **#end**

Usage:

- *condition* - If a boolean, considered true if it has a true false; if not a boolean, considered true if not null.
- *output* - May contain VTL.

Examples:

- Equivalent Operator: #if($foo == $bar)
- Greater Than: #if($foo > 42)
- Less Than: #if($foo < 42)
- Greater Than or Equal To: #if($foo >= 42)
- Less Than or Equal To: #if($foo <= 42)
- Equals Number: #if($foo = 42)
- Equals String: #if($foo = "bar")

#foreach - Loops through a list of objects

Format:

#foreach($ref1 **in** $ref2 **)** [statement...] **#end**

Usage:

- *$ref1* - The first variable reference is the item.
- *$ref2* - The second variable reference is the list that holds the items.
- *statement* - What is output each time Velocity finds a valid item ($ref1) in the list ($ref2).

Velocity provides an easy way to get the loop counter so that you can do something like the following:

```
<table>
#foreach( $customer in $customerList )
    <tr><td>$velocityCount</td><td>$customer.Name</td></tr>
#end
</table>
```

The default name for the loop counter variable reference, which is specified in the velocity.properties file, is $velocityCount. By default the counter starts at 1, but this can be set to either 0 or 1 in the `velocity.properties` file. Here's what the loop counter properties section of the `velocity.properties` file appears:

```
# Default name of the loop counter
# variable refernce.
counter.name = velocityCount
# Default starting value of the loop
# counter variable reference.
counter.initial.value = 1
```

#include - Renders local file(s) that are not parsed by Velocity

Format:

#include(arg[, arg2, ... argn] **)**

- *arg* - Refers to a valid file under TEMPLATE_ROOT.

Examples:

- String: #include("disclaimer.txt", "opinion.txt")
- Variable: #include($foo, $bar)

#parse - Renders a local template that is parsed by Velocity

Format:

#parse(arg)

- arg - Refers to a template under TEMPLATE_ROOT.

Examples:

- String: #parse("lecorbusier.vm")
- Variable: #parse($foo)

Recursion permitted. See parse_directive.maxdepth in velocity.properties to change from parse depth. (The default parse depth is 10.)

#stop - Stops the template engine

Format:

#stop

Usage:

This will stop execution of the current template. This is good for debugging a template.

#macro - Allows users to define a Velocimacro (VM), a repeated segment of a VTL template, as required

Format:

#macro(vmname $arg1[, $arg2, $arg3, ... $argn] **)** [VM VTL code...] **#end**

- *vmname* - Name used to call the VM (*#vmname*)
- *$arg1 $arg2 [...]* - Arguments to the VM. There can be any number of argumentss, but the number used at invocation must match the number specified in the definition.
- *[VM VTL code...]* - Any valid VTL code, anything you can put into a template, can be put into a VM.

Once defined, the VM is used like any other VTL directive in a template.

```
#vmname( $arg1 $arg2 )
```

VMs can be defined in one of two places:

1. *Template library:* can be either VMs pre-packaged with Velocity or custom-made, user-defined, site-specific VMs; available from any template
2. *Inline:* found in regular templates, only usable when *velocimacro.permissions.allowInline=true* in `velocity.properties`.

6.3 Comments

Comments
Comments are not rendered at runtime. Single Line Example: ## **This is a comment.**   Multi-line Example: #* **This is a multiline comment.** **This is the second line** *#

References:

Ja-Jakarta Project Velocity Users Guide, HTTP://www.jajakarta.org., *Copyright © 1999-2002, The Apache Software Foundation*

Ja-Jakarta Project VTL Reference, HTTP://www.jajakarta.org., *Copyright © 1999-2002, The Apache Software Foundation*

EnterWorks Users Guide, HTTP://www.enterworks.com., *Copyright © 2010, EnterWorks Inc.*