



## Lesson 5: Macros

### Learning Objectives

- Understand:
  - What macros are
  - The difference between inline macros and library macros
- Learn how to:
  - Create a macro
  - Use a macro
  - Use a library macro

[Collapse all](#)

#### Understanding Code Validation

---

- It is difficult to find a good place for the discussion of code validation, because it is related to a few different topics
- I put the discussion here in this lesson because I need to talk about macros, and macro code is always validated
- A conditional structure like `#if` controls the execution of code
- When a block of code appears in a conditional that is always evaluated to `false`, we may think that the block of code is never executed
- Now there is a subtle difference between code execution and code validation
- Code validation is a procedure in which the code is processed no matter where it appears (I am not talking about a comment though)
- Macro code is always validated; that is to say, it does not make sense to wrap a macro definition within a conditional: the conditional is always ignored and the macro code is always processed
- Consider the following:

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

#myMacro
```

- Since the macro definition is wrapped inside a conditional that is always evaluated to false , we may think that the macro is not defined
- And yet, this is not the case; the macro is defined
- The reason behind this is that the Velocity engine always validates any macro definition code, no matter where it appears
- By validation, I mean that the Velocity engine checks to see if a block of code is well-formed to avoid syntax errors, lexical errors, etc.
- The process of code validation also checks if the code involved throws an exception
- By the process of validation, the macro is hence defined
- To see how this works, try to remove the closing parenthesis in `#macro( myMacro )`
- However, validation is not identical to execution
- To see the difference, consider the following:

```
#if( false )
#set( $变量 = "dash" )
#end

$变量
```

Here we try to use Unicode characters for the name of a variable, and because this is a lexical error and the code fails validation, we cannot submit the code.

- Now change the variable name so that the code becomes valid:

```
#if( false )
#set( $dash = "dash" )
#end

$dash
```

We can now submit the code. But note that the variable `$dash` is not defined. That is to say, the `#set` directive, unlike `#macro` , is validated but not executed.

- In [Lesson 3: Variables](#) I mentioned that `$_PropertyTool.isNull` is not a reliable method
- Now we can elaborate why it is not: the code wrapped inside a conditional using `$_PropertyTool.isNull( $x )` (or `!$_PropertyTool.isNull( $x )`) as the boolean expression is validated; it does not matter whether `$_PropertyTool.isNull( $x )`

returns true or not

- If the code throws an exception, then we cannot submit it due to code validation, even though the code is logically sound
- To avoid this problem caused by exception throwing, we have to use a normal conditional without involving `$_PropertyTool.isNull`
- One easy way to implement this is to use `#if( $x.Class.Name )` instead; the code within the conditional can throw any exception, and yet it won't cause any problem upon code submission
- For more discussion, see [Lesson 3: Numbers, Strings, Dates, XML, and XSLT](#)

---

## Parameters vs. Arguments

---

- In these courses I maintain a distinction between two terms when talking about macros: **parameter** and **argument**
- A parameter is a local variable created in the context of a definition of a macro; it is a place-holder for a value
- An argument is the actual object or value passed into an invocation of a macro, to be stored in a corresponding parameter
- A parameter is a variable listed among other parameters within the parentheses enclosing the macro name and all parameters in the definition, whereas an argument is the value passed in to take the place of a parameter when the macro is invoked
- Example:

```
#macro( myMacro $var )  
    $var  
#end  
  
#myMacro( 4 )
```

In this example, the macro `#myMacro` has one parameter, namely, `$var`, and the macro outputs the value of the parameter. In `#myMacro( 4 )`, the value 4 is passed in as an argument, and this argument, when taking the place of the parameter `$var`, will be output.

---

## What is a Macro?

---

- A macro is a block of reusable code associated with a name, defined by using the `#macro` directive

- A macro can be called, or invoked, to perform some procedures
- When a macro is invoked, the macro name is preceded by the `#` character
- A macro can accept a list of zero or more parameters
- Parameters of a macro are local variables
- If a macro does not accept any arguments, then the invocation of the macro can be either `#myMacro` or `#myMacro()` with a pair of parentheses
- Even if a macro can take parameters, when it is invoked, there can be no arguments passed in; in this case, parameters within the macro are undefined when the macro is invoked
- A macro cannot return any values
- A macro can contain other macros
- A macro can be overwritten by another macro of the same name

---

## Creating a Macro

---

- Use the following structure to create a macro:

```
#macro( macroName $param1 $param2 ... $paramN )  
    ## macro code here  
#end
```

- `#macro` is a directive
- Following this directive, there should be a pair of parentheses
- Within the parentheses, there should be at least one item, the name of the macro to be declared
- Following the macro name, there can be a list of parameters
- Note that there are no punctuation marks nor the `#` character inside the parentheses
- If there are  $n$  items in the parentheses, then the macro defined will take  $n - 1$  arguments when invoked
- `#end` marks the end of the definition

---

## Using a Macro

- To use a macro, follow this pattern:

```
#macroName( $param1 $param2 ... $paramN )
```

- The macro name is preceded by #
- Following the macro name, there is a pair of parentheses
- Inside the parentheses, put in a list of arguments: variables or values
- The number and order of the arguments should correspond to the parameter list in the definition of the macro
- Items that can appear inside parentheses as arguments:
  - Variables pointing to any Java objects
  - String literals
  - Number literals
  - Lists
  - Maps
  - true and false
- No punctuation marks inside the parentheses
- No operators like + inside the parentheses
- If the macro does not accept any parameters, then the pair of parentheses are optional
- Even if a macro is defined to take parameters, when it is invoked, there can be no arguments, or less arguments than defined, that are passed in
- When there are less arguments than defined passed in, arguments are paired with their corresponding parameters in the macro definition, and they are paired from left to right; the last or last few parameters will have no value(s)

---

## Detecting Argument Types

---

- When designing a macro that has parameters, for a specific parameter, we may expect value of a specific type, or we may want to allow for objects of different types to be passed in
- For example, here is a skeleton of a macro:

```
#macro( chanwDisplayObjClass $obj )  
    ## code  
#end
```

We want to display some information of the object passed in. The type of the `$obj` argument can be a `String` (`java.lang.String`), a `java.lang.Class` object, or an object of any other type.

- We will want to test the type of the argument passed in, using `#if...#elseif...#else` to determine what to do with a specific type
- To recover the class name of an object, use `$x.Class.Name`
- Therefore, we can have code like the following:

```
#macro( chanwDisplayObjClass $obj )
  #set( $objString = "" )
  #set( $objType   = $obj.Class.Name )

  #if( $objType == "java.lang.String" )    ## class name, a String
    #set( $objString = $obj )
  #elseif( $objType == "java.lang.Class" ) ## Class obj
    #set( $objString = $obj.Name )
  #else                                     ## obj of other types
    #set( $objString = $obj.Class.Name )
  #end

  ## code
#end
```

Here we create a global variable `$objString` and assign the empty String to it. Then we use `#if...#elseif...#else` to detect the type of the argument object. We want to find the type String of the object and store it in `$objString`. After that, we can use the value of the type String to determine what to do next.

---

## Supplying Default Values

---

- When a macro is defined with a parameter list, whenever it is possible, it is a good idea to supply default values for these parameters
- By supplying default values for parameters, we can avoid terminating code execution when unacceptable values are passed in or when there are missing arguments
- Assuming that we want a parameter to store an integer, we can first check to see if the parameter is defined (storing a value)
- Next we check if the type of the value is `java.lang.Integer`
- If the value is of the right type, we assign the value to a global variable
- If there is no value passed in, or if the value is not of the right type, we assign a suitable default value (e.g., 0 or 1) to the global variable
- From this point on, we only use the global variable in the macro

- Example:

```
#macro( myMacro $num )
  #if( !$num.Class.Name || $num.Class.Name != "java.lang.Integer" )
    #set( $myDefault = 0 )
  #else
    #set( $myDefault = $num )
  #end

  ## consume $myDefault
#end

## a shorter version
#macro( myMacro $num )
  #set( $myDefault = $num )

  #if( !$num.Class.Name || $num.Class.Name != "java.lang.Integer" )
    #set( $myDefault = 0 )
  #end

  ## consume $myDefault
#end
```

---

## Interaction between Global and Local Variables

---

- When checking the validity of input values passed into a macro, watch out for the unintended interaction between global and local variables
- When there is no value passed in for a specific parameter, the macro should either supply a default value for the parameter or stop the execution
- However, if the name of the local parameter is identical to the name of a global variable, then the global variable will take the place of the local variable because the local variable is undefined and the value of the global variable is visible to the macro
- Testing the value passed in may fail due to the dual facts that the local variable is undefined and that there is a global variable of the same name
- If a local variable and a global variable of the same named are both defined, then within the macro, only the local variable is available, and the global one is said to be overshadowed by the local one
- Collision of names may seem unlikely if we use two different naming conventions, one for global variables (like using certain prefixes), the other for local variables
- However, reusable macro may have local variable names like `$str` that may collide with names of some global variables defined by the client code
- To avoid the collision of names, make sure that even local variables have unique names

- One possible solution: using the macro name as part of the variable names

---

## Maps as Arguments

---

- `HashMap` objects can be passed into macros as arguments
- Strings can be used as keys
- Advantages of using maps as arguments:
  - To avoid long lists of parameters when defining macros
  - No need to worry about the order and optionality of parameters
  - As long as the name of the parameter for the map is protected from collision, any Strings can be used as keys
- When a macro that accept a map as an argument is defined, the macro must take care of optionality of values provided; default values should be supplied when optional values are missing
- Thorough documentation is required for reusable macros

---

## Returning Values

---

- A macro is not a function and cannot return a value
- But there are still ways to mimic a function and "return" a value from a macro
- We can use global variables
- The first way is to use a global variable like a drop box; inside the macro, we assign the returned value to this global variable:

```
#macro( myMacro )  
  ## code  
  
  #set( $globalResult = $returnValue )  
#end  
  
## before invoking the macro  
#set( $globalResult = "" )  
  
## invoke the macro  
#myMacro()
```

Note that the macro contains the global variable. This means that when the macro is defined, the global variable must be available first.



- The second way is to create a global variable right inside the macro:

```
#macro( myMacro )
  ## code

  #set( $myMacro = $returnedValue )
#end

## invoke the macro
#myMacro

## consume the returned value
$myMacro
```

Here I recommend using the same name for the macro and the variable to store the returned value. Code using the macro will use the same name as a variable name to access the returned value.

- If a macro outputs one or more Strings, then these Strings can be treated as some kind of returned values when the macro is invoked in a specific way
- Example:

```
#macro( myMacro )
  Hey!
  #set( $dummy = "" )
  Hello, World!
  #set( $myMacro = 3 )
#end

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

- The macro outputs two Strings: Hey! and Hello, World!
- It also creates two global variables \$dummy and \$myMacro
- The variable \$dummy is introduced to break up the two Strings
- The line #set( \$result = "#myMacro" ) invokes the macro and stores the returned String in the variable \$result
- Note how the invocation appears inside double quotes and the result is stored in \$result
- When the macro is invoked inside double quotes, the Strings Hey! and Hello, World! will not be output; instead, they are concatenated and stored
- We have also stored a separate returned value in the global variable \$myMacro
- Any macro can be invoked by using this double-quote method
- Whether there is a String stored in the variable \$result depends on whether the macro outputs any String(s)

- The double-quote method only stores a String in the variable `$result`, provided the macro outputs some String(s)
- If the macro outputs an object or a numeric literal, then the value will be turned into a String:

```
#macro( myMacro )
  $_PropertyTool  ## output the object
  3               ## output the number 3
#end

#set( $result = "#myMacro" )
$result
$result.Class.Name
```

And the snippet outputs the following:

```
com.hannonhill.cascade.velocity.PropertyTool@41af8d29
3

java.lang.String
```

Note how the address of `$_PropertyTool` and the String "3" are concatenated and stored in the variable `$result`.

- Because the variable `$result` stores a String, we can use `java.lang.String` methods like `replaceAll` to manipulate the String before outputting it:

```
#set( $result = "#myMacro" )
$result.replaceAll( "\s+", " " )
```

The line `$result.replaceAll( "\s+", " " )` eliminates all extra whitespace characters in the String before outputting it.

- There is a third way to store a returned value: the client code can supply a name out of which a global variable can be created and assigned the return value
- The variable name must be passed into the macro as a parameter
- We must use `#evaluate` to create this new variable (see [Lesson 2: #define and #evaluate](#))

---

## Using `$bodyContent`

---

- Inside a macro, a local variable named `$bodyContent` is always created
- The type of the value of this variable is `org.apache.velocity.runtime.directive.Block$Reference`
- This object only has a `toString` method
- We have seen the following two different ways to invoke a macro:

```
#macro( myMacro )  
    #set( $myMacro = 3 )  
#end  
  
#myMacro                ## with or without ()  
#set( $result = "#myMacro()" )  ## use double quotes
```

- There is another way to invoke a macro:

```
#macro( myMacro )  
    #set( $myMacro = 3 )  
#end  
  
#@myMacro()#end
```

A few things about this way of invoking a macro:

- There is a @ character between # and the macro name
- Parentheses must be used
- #end must be used
- There can be a value between the closing parenthesis and #end ; the value can be any object
- When there is an object inserted in the invocation, the returned value of the toString method of the object becomes the returned value of the toString method of \$bodyContent
- When the macro is invoked by #@myMacro()#end , \$bodyContent.toString() returns the empty String
- We can put in a String in the invocation like this:

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

Now the variable \$bodyContent.toString() will return the String Hello, World!

- The macro itself can accept parameters:

```
#macro( myMacro $var )  
    #set( $myMacro = $var )  
#end  
  
## invoke the macro  
#@myMacro( 3 )Hello, World!#end
```

- The String value stored in \$bodyContent acts like an extra argument passed into the macro
- We can also invoke a macro by combining #@ and double quotes:

```
#macro( myMacro )
    $bodyContent.toString()
#end

#set( $result = "#@myMacro()Hello#end" )
$result
```

---

## Why Macros?

---

- To promote reusability
- To modularize code
- To better organize code
- To separate page content processing from actual output

---

## Using Library Macros

---

- One of the purposes of creating macros is to promote reusability
- A macro can be created inline, meaning that the macro resides in the same Velocity format where the macro is also invoked
- For code reusability, it is better to separate the macro and its invocation into separate formats; that is to say, we can put the macro in a library format, and invoke it in client formats
- Important concepts related to reusability:
  - A library format contains only macros, which are called library macros; a library macro is normally not invoked in a library format, unless the invocation appears in another library macro or is invoked to set up some global utility objects
  - A standalone invocation of a library macro (i.e., an invocation not appearing in a macro) is called an entry point to the library macro of the library format
  - A library format does not contain any entry points, and when such a format is attached to a region, for example, it won't do anything
  - Therefore, we must provide entry points to access library formats and library macros
  - A format that provides entry points to a library format is a client format of the library format
- To access a library format in a client format, use the `#import` directive:

```
#import( "site://_common_assets/formats/library/velocity/chanw-global-stack" )
```

- Site name is optional if the library format and client format reside in the same site; if the site name is not used, then we will have:

```
#import( "/formats/library/velocity/chanw-global-stack" )
```

- Once a library format is imported, all macros defined in the library format can be invoked
- Global variables created in the library format are also accessible
- We still need to invoke some macro or other in the client format; that is, we have to create some entry points to the library

---

## Modularizing Code

---

- Reusability is not the only reason why we want to define macros
- When we deal with extremely long code, we may want to group blocks of code into macros for better organization and division of labor
- Here is an example:

```
#macro( createHtml )  
<html $siteConfigMap['htmlAttributes']>  
  ***##createHead  
  ***##createBody  
</html>  
#end
```

- We can surely write all the page-rendering code instead of invoking the macro named `createHtml`; but by defining this macro, we can split the code into two parts, one for processing the `head` element, the other for processing the `body` element
- When processing the `body` element, we can go further:

```
#macro( createBody )  
<body $siteConfigMap['bodyAttributes']>  
  #createHeader  
  #createMain  
  #createFooter  
</body>  
#end
```

- By so doing, blocks of code will become smaller and easier to handle
- Better division of labor

## Organizing Code

---

- Macros can be defined and organized in some special way to show dependencies
- In the Standard Model, when a data definition block is attached to a block chooser, a macro is invoked automatically to process the block data
- The automatically invoked macro can contain related presentational macros for different designs
- Example:

```
#macro( processAccordionGroup $map )
    #chanwInvokeMacro( "${design_namespace}ProcessAccordionGroup" [ "map" ] )

    #macro( rwd4ProcessAccordionGroup $map )
        ## code to produce an accordion for design rwd4
    #end

    #macro( ucoCreateAccordion $map )
        ## code to produce an accordion for design uco
    #end
#end
```

- To understand what is going on in this example, consider the following facts first:
  - A site is associated with a design
  - The value of the design String is stored in the variable `$design_namespace`
  - When the macro named `processAccordionGroup` is invoked, the variable `$design_namespace` is visible and should store a value like `rwd4`
  - When `#chanwInvokeMacro( "${design_namespace}ProcessAccordionGroup" [ "map" ] )` is invoked by `processAccordionGroup`, the String `"${design_namespace}ProcessAccordionGroup"` should be resolved to a macro name like `rwd4ProcessAccordionGroup`
  - Then the macro named `rwd4ProcessAccordionGroup` is invoked to produce an accordion for the design named `rwd4`
- The macros named `rwd4ProcessAccordionGroup` and `ucoProcessAccordionGroup` can be put anywhere; yet they are defined right inside `processAccordionGroup` to show that there is a close dependency of `processAccordionGroup` on these two macros
- Macros defined inside other macros can still be invoked independently

---

## Separating Processing from Outputting

- The `head` element (the first child of `html`) should be output before the `body` element
- Yet there may be occasions where we need to process the `body` element before the `head` element
- For example, we may want to insert `link` elements into the `head` element when some blocks are attached to the page
- Instead of directly outputting the HTML markups of the `body` element, we can store them in a variable, using the double quote notation to invoke a macro:

```
#set( $bodyContent = "#createBody" )  
#createHead  
$bodyContent
```

- Processing code must be wrapped inside a macro to enable this separation of processing from outputting
- Because the `body` element is processed before the `head` element, the processing of `body` can trigger insertion of contents into `head`

---

## Recursive Macros

---

- In programming, a recursive function is a function that:
  - Directly calls itself (immediate recursion); or
  - Eventually calls itself through a cyclic function calling path
- Since a recursive function directly or indirectly calls itself, this may lead to an infinite chain of function calls
- For a recursive function to work properly, normally three conditions should be met:
  - There must be a stopping condition; when the condition is met, the recursion stops
  - There should be a base case, when the stopping condition is met, and in the base case the function is not called
  - The reduction of some parameter so that eventually the stopping condition will be met
- Good examples of using recursions are summation, factorial, and the Fibonacci series
- A macro can be recursive; example:

```
#recursiveMacro( 20 )

#macro( recursiveMacro $num ) ## positive integers only
  $num
  #if( $num > 1 )
    #set( $minusOne = $num - 1 )
    #recursiveMacro( $minusOne )
  #end
#end
```

- In this example, both the base case and stopping condition is when `$num` stores the value 1
- `#set( $minusOne = $num - 1 )` ensures that the parameter is reduced in every recursive loop
- In Velocity, by default recursion is capped to 20; therefore, `#recursiveMacro( 21 )` is not allowed
- Normally, we do not want to use recursion to compute numbers like summation because recursion is very inefficient
- But it is instructive to try to see how recursion works
- Consider the following:

```
#summation( 4 )
$summation

#macro( summation $num ) ## positive integers only
  #if( $num > 1 )
    #set( $minusOne = $num - 1 )
    #summation( $minusOne )
    #set( $summation = $num + $summation )
  #else
    #set( $summation = $num )
  #end
#end
```

Here we are trying to use the recursive macro `#summation` to find the answer 10; i.e.,  $1 + 2 + 3 + 4$

- The output from the format is actually 7
- To see what goes wrong, we can add some debugging code:



```

#summation( 4 )
$summation

#macro( summation $num ) ## positive integers only
    Calling macro with $num
    #if( $num > 1 )
        Value of num: $num
        #set( $minusOne = $num - 1 )
        Value of minusOne: $minusOne
        #summation( $minusOne )
        Value of minusOne after macro call $_EscapeTool.getH()sumation( $minusOne ):
        #set( $summation = $num + $summation )
        Value of num: $num
        Value of summation: $summation
    #else
        #set( $summation = $num )
    #end
#end

```

- This is the formatted output:

```

Calling macro with 4
Value of num: 4
Value of minusOne: 3

Calling macro with 3
Value of num: 3
Value of minusOne: 2

Calling macro with 2
Value of num: 2
Value of minusOne: 1

Calling macro with 1
Value of minusOne after macro call #sumation( 1 ): 1
Value of num: 1
Value of summation: 2
Value of minusOne after macro call #sumation( 1 ): 1
Value of num: 1
Value of summation: 3
Value of minusOne after macro call #sumation( 1 ): 1
Value of num: 4
Value of summation: 7

7

```

- Note the following:
  - The actual adding of numbers begins only when `#summation( 1 )` is invoked
  - When `#summation( 1 )` is invoked, the global variable `$minusOne` stores 1
  - Since `$minusOne`, a global variable, consistently stores 1 when we need its values

(when we call `#summation( $minusOne )`), we are adding  $1 + 1 + 1 + 4$  instead of  $1 + 2 + 3 + 4$

- Conclusion: global variables can be contaminated by recursion
- We may want to stick with the variable `$num` without introducing another global variable:

```
#summation( 4 )
$summation

#macro( summation $num ) ## positive integers only
  #if( $num > 1 )
    #set( $num = $num - 1 )
    #summation( $num )
    #set( $summation = $num + $summation )
  #else
    #set( $summation = $num )
  #end
#end
```

- This time the output is 4 and we know that this is because the variable `$num` becomes a global variable when `#set( $num = $num - 1 )` is executed and the variable is contaminated
- To avoid the contamination, we need an independent data structure to store the value of `$num`; we can use the global stack
- Consider the following:

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

#summation( 10 )
$summation

#macro( summation $num ) ## positive integers only
  #if( $num > 1 )
    #chanwPushGlobalStack( $chanwGlobalStack $num )
    #set( $minusOne = $num - 1 )
    #summation( $minusOne )
    #chanwPopGlobalStack( $chanwGlobalStack )
    #set( $summation = $chanwPopGlobalStack + $summation )
  #else
    #set( $summation = $num )
  #end
#end
```

- In every loop of the recursion, the value of `$num` is pushed into the stack
- The value of `$minusOne` is recalculated based on the popped value of `$num`
- Both values of the two variables are protected from contamination
- **Important:**

- When `#set` is used, we are introducing a global variable
- Global variables will be contaminated within a recursive loop when its value is modified
- A local variable created by the macro becomes a global one if we use `#set` to change its value; local variables should never be modified
- To keep track of values for each recursion, we may need to use a global data structure like a stack
- When do we want to use recursion: processing a folder
  - A folder can contain folder descendants
  - The folder hierarchy rooted at the starting folder is a tree
  - When processing folders, we first process the current folder, and then we check if the current folder has any folder children; the stopping condition of any branch is met when we reach the terminal folder that does not contain any folder children
  - The base cases are when we process folder-contained assets other than folders
  - When we traverse from a parent folder to a child folder, we are getting closer and closer to a terminal folder; there is no need of reduction of any parameter because the reduction is guaranteed in a finite tree
  - This recursive traversal of a tree is an example of depth-first traversal
  - The traversal is still capped at a depth of 20
  - To avoid the recursion cap, we should use iteration (breadth-first traversal)
  - Another way to avoid recursion is to represent the folder structure using a Java tree

---

### Finding the $n$ th Term of the Fibonacci Series

---

- To conclude this lesson, I want to look at a macro that can be used to find the  $n$ th term of the Fibonacci series
- The Fibonacci series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Specifications:
  - The name of the macro is `#fibonacci`
  - It takes a positive integer as its argument; this is the value of  $n$
  - It returns the  $n$ th term of the series; the returned value is stored in a variable named `$fibonacci`
  - The client code can simply contain two lines of code:

```
#fibonacci( 10 )  
$fibonacci ##=> 55
```

- The macro must be recursive
- To find `#fibonacci( n )`, we must first find `#fibonacci( n - 1 )` and `#fibonacci( n - 2 )`
- To appreciate what is involved, let us look at a PHP function with the same functionality:

```
function fibonacci( $term )  
{  
    if( $term == 1 || $term == 2 )  
    {  
        return 1;  
    }  
    else  
    {  
        return fibonacci( $term - 1 ) + fibonacci( $term - 2 );  
    }  
}
```

- Differences between the PHP function and our macro:
  - A macro does not return anything; at most, we can store the desired returned value in a global variable
  - Values resulted from an operation like `$term - 1` must be stored in global variables; an expression like `$term - 1` cannot be used with a macro invocation directly
- Values we need to keep track of in the recursive macro:
  - The value of `$term`
  - The value of `$term - 1`
  - The value of `$term - 2`
  - The value of `$fibonacci`
- **Important:**
  - The local variable `$term` should never be modified in the macro; this is the only value we can rely on
  - We will need a global data structure to store other values
  - Since many different global data structures can be used, the macro can be implemented in many different ways
- Here I will use a map
- Since the variable `$term` is local and its value is never modified in the macro, we can use

its value as a key in each recursion

- A `$term` value is associated with an inner map, storing three items: `minusOne`, `minusTwo`, and `answer`
- In each recursion, we update all three values, if they are not defined, but we never overwrite existing values
- Here is the macro:

```
#macro( fibonacci $term )
  ## initial setup
  #set( $ANSWER = "answer" )
  #set( $MINUS_ONE = "minusOne" )
  #set( $MINUS_TWO = "minusTwo" )
  #set( $TEMP = "temp" )
  ##
  #if( !$fibonacciMap.Class.Name )
    #set( $fibonacciMap = {} )
  #end
  ## inner map
  #if( !$fibonacciMap[ $term ].Class.Name )
    #set( $fibonacciMap[ $term ] = {} )
  #end
  ##
  #if( $term == 1 || $term == 2 )
    #if( !$fibonacciMap[ $term ][ $ANSWER ].Class.Name )
      #set( $fibonacciMap[ $term ][ $ANSWER ] = 1 )
    #end
  #else
    ## $term is not global and never changes
    #set( $minusOne = $term - 1 )
    #set( $minusTwo = $term - 2 )
    ## store $minusOne and $minusTwo for the current $term
    #if( !$fibonacciMap[ $term ][ $MINUS_ONE ].Class.Name )
      #set( $fibonacciMap[ $term ][ $MINUS_ONE ] = $minusOne )
    #end
    #if( !$fibonacciMap[ $term ][ $MINUS_TWO ].Class.Name )
      #set( $fibonacciMap[ $term ][ $MINUS_TWO ] = $minusTwo )
    #end
    ## find the answer of $term
    #if( !$fibonacciMap[ $term ][ $ANSWER ].Class.Name )
      Invoking macro with $fibonacciMap[ $term ][ $MINUS_ONE ]
      #fibonacci( $fibonacciMap[ $term ][ $MINUS_ONE ] )
      #set( $fibonacciMap[ $term ][ $TEMP ] = $fibonacci )
      Invoking macro with $fibonacciMap[ $term ][ $MINUS_TWO ]
      #fibonacci( $fibonacciMap[ $term ][ $MINUS_TWO ] )
      #set( $fibonacciMap[ $term ][ $ANSWER ] = $fibonacci + $fibonacciMap[ $term ][ $TEMP ] )
    #end
  #end
  #set( $fibonacci = $fibonacciMap[ $term ][ $ANSWER ] )
#end
```

- Here is the test code:

```
#import( 'site://_brisk/core/library/velocity/chanw/chanw-date-time-utilities' )  
$chanwStartMeasurement  
#fibonacci( 30 )  
  
$fibonacci  
Time taken: $chanwEndMeasurement milliseconds
```

- Here is the output:


Invoking macro with 29  
Invoking macro with 28  
Invoking macro with 27  
Invoking macro with 26  
Invoking macro with 25  
Invoking macro with 24  
Invoking macro with 23  
Invoking macro with 22  
Invoking macro with 21  
Invoking macro with 20  
Invoking macro with 19  
Invoking macro with 18  
Invoking macro with 17  
Invoking macro with 16  
Invoking macro with 15  
Invoking macro with 14  
Invoking macro with 13  
Invoking macro with 12  
Invoking macro with 11  
Invoking macro with 10  
Invoking macro with 9  
Invoking macro with 8  
Invoking macro with 7  
Invoking macro with 6  
Invoking macro with 5  
Invoking macro with 4  
Invoking macro with 3  
Invoking macro with 2  
Invoking macro with 1  
Invoking macro with 2  
Invoking macro with 3  
Invoking macro with 4  
Invoking macro with 5  
Invoking macro with 6  
Invoking macro with 7  
Invoking macro with 8  
Invoking macro with 9  
Invoking macro with 10  
Invoking macro with 11  
Invoking macro with 12  
Invoking macro with 13  
Invoking macro with 14  
Invoking macro with 15  
Invoking macro with 16  
Invoking macro with 17  
Invoking macro with 18  
Invoking macro with 19  
Invoking macro with 20  
Invoking macro with 21  
Invoking macro with 22  
Invoking macro with 23  
Invoking macro with 24  
Invoking macro with 25

```
Invoking macro with 26  
Invoking macro with 27  
Invoking macro with 28  
  
832040  
Time taken: 34 milliseconds
```

- Note that even though the macro is recursive, answers are stored, and hence cached, to reduce execution time

---

## Examples

- [introductory/05\\_macros](#) 

## References

- [Different ways to invoke macros](#) 