
Adobe
CFML Features for More Modern Coding

CFML Features for More Modern Coding

Dan Fredericks

fmdano@gmail.com

@fmdano74

http://www.meetup.com/nvcfug/

Presentation link:

What this Presentation will cover:

- CFScript Support
- QueryExecute() Function
- Member Functions
- Elvis Function
- Closures
- Map and Reduce
- First Class Functions and Callbacks
- CF 12 Special

CFScript

- Seems like first CFScript was implemented in CF 4.0
  - Tags seemed ok back then because pre CFC's (CF Components)
    - Used UDF's and Custom Tags to reuse and separate code
  - CF MX (6.0) brought about CFC's so separation of code
    - CFCs - you create methods, which are ColdFusion user-defined functions, in the component page. You pass data to a method by using parameters. The method then performs the function and, if specified in the cfreturn tag, returns data.
  - CF 9 added script based cfc support
    - You could write a full CFC with cfscript and not need to include code in cfscript block
- CF11 adds full Script Support
  - most of the cfscript example will be CF 11 unless otherwise noted

CFScript

- Community has different opinions on using Tag vs Script
  - Using MVC Tags on View, Script on Controller and Model
  - Script syntax is cleaner
  - Closer to other ECMA script languages
  - Lots of tag syntax is more characters
  - I am more comfortable with Tags
  - Documentation is lacking for Script (check *cfdocs.org*)
  - Tags were superior for SQL integration (*talk about this later*)
  - Newer features such as Closures are script based
- Do what is best for you but try to be consistent!!!

CFScript

Syntax – tags without Bodies:

cfTagName(attribute-pairs*);

writeDump(var);

writeOutput(expression);

Location(“mypage.cfm”,”false”);

Syntax – tags with Bodies:

cfTagName(attribute-pairs*)
{ …};

cfDocument(format=“PDF”)
{ //code }

For (i=1; i<=10; i++) {
  //code
}

CFScript

- There are some tags that have multiple implementations
  - These implementations are due to old syntax or dual syntax (CF9 cfc's)

```
<cfscript>
//CF9 syntax
thread action="run" name="testName" {
        thread.test = "CFML";
}
//CF11 syntax
cfthread( action="run" name="testName"){
        thread.test = "CFML";
}
<cfscript>
```

CFScript

```
<cfscript>
//CF9 syntax
param name=“local.summit” type = “string” default = “CFSummit”;

//CF11 syntax
cfparam( name = “local.summit” type = “string” default = “CFSummit”);
</cfscript>
```

CFScript

```
<cfscript>
//CF9 syntax
abort “some message”;

//CF11 syntax
cfabort( showError = “some message”);
</cfscript>
```

CFScript

Tag Example

- Loop

```
<cfloop from="1" to="10"index="i">
        <cfoutput>#i#</cfoutput>
</cfloop>
```

Script Example

- For Loop

```
for (i=1; i <=5; i++) {
   // all statements in the block are looped over
   result = i * 2;
   writeOutput(result);
}
```

- While Loop

```
while (condition) {
   // statements
}
```

CFScript

Tag Example

- If/elseif/else example:

```
<cfset count = 10>
<cfif count GT 20>
  <cfoutput>#count#</cfoutput>
<cfelseif count GT 8>
   <cfoutput>#count#</cfoutput>
<cfelse>
   <cfoutput>#count#</cfoutput>
<cfif>
```

Script Example

- If/elseif/else example:

```
<cfscript>
count = 10;
 if (count > 20) {
   writeOutput(count);
 } else if (count == 8) {
   writeOutput(count);
 } else {
   writeOutput(count);
 }
</cfscript>
```

CFScript

Tag Example

- Query Loop

```
<cfset platform = ["Adobe ColdFusion", "Railo", "Lucee"]>
<cfset myQuery = queryNew(" ")>
<cfset queryAddColumn(myQuery, "platform", "CF_SQL_VARCHAR",
      platform)>
<cfloop index="i" from="1" to="#myQuery.recordCount#">
      <cfoutput><li>#myQuery["platform"][i]#</li></cfoutput>
</cfloop>
```

OR

```
<cfloop query="myQuery" group="platform">
      <cfoutput><li>#platform#</li></cfoutput>
</cfloop>
```

Script Example

- Query Loop

```
q = queryNew("id,data", "integer,varchar",
                  [  [11, "aa"], [22, "bb"], [33, "cc"]  ] );
for (row in q){
    writeOutput("#q.currentRow#:#row.id#:#row.data#;");
    //result:   1:11:aa;2:22:bb;3:33:cc;
}
```

OR

```
cfloop(query=q, group="fk"){
   writeOutput("<strong>#fk#</strong>");
}
```

CFScript

Tag Example

```
<cfcomponent displayname="utils" output="false" >
    <cfproperty name="version" type="string" default="0.0.1" >
    <cffunction name="doReverse" access="public" returntype="string" hint="I reverse the supplied string" >
        <cfargument name="stringToReverse" required="true" type="string" >
        <cfreturn reverse(arguments.stringToReverse) >
    </cffunction>
    <cffunction name="reverseArrayOrder" access="public" returntype="array" hint="I reverse an array's order" >
        <cfargument name="arrayToReverse"  type="array" default='["Adobe ColdFusion", "Lucee", "Railo"]' >
        <cfset local.result = '' />
        <cfloop array="arguments.arrayToRevers" index="local.a" >
            <cfset arrayAppend(local.result, local.a) >
        </cfloop>
        <cfreturn local.result />
    </cffunction>
</cfcomponent>
```

CFScript

Script Example

```
component displayname="Utils" output="false" {
   (property name="version" type="string" default="0.0.1";) - not really needed

    public string function doReverse(required string stringToReverse)
       hint="I reverse the supplied string"
    {
       return reverse(arguments.stringToReverse);
    }

    public array function reverseArrayOrder(array arrayToReverse = ["Adobe ColdFusion", "Lucee", "Railo"])
       hint="I reverse an array's order"
    {
       var result = [];
       for (var i = arrayLen(arguments.arrayToReverse); i >= 1; i--)
       {
          result.append(arguments.arrayToReverse[i]);
       }
       return result;
    }
}
```

CFScript

Resources:

- ColdFusion Online Docs : https://helpx.adobe.com/coldfusion/cfml-reference/coldfusion-tags/tags-r-s/cfscript.html#cfscript-Scriptsupportfortags
- CFDocs: http://cfdocs.org/
- CFChef GitHub: https://github.com/cfchef/cfml-tag-to-script-conversions/blob/master/README.md#general
- Adam Cameron GitHub: https://github.com/adamcameron/cfscript/blob/master/cfscript.md#database
- Pete Freitag Cheat sheet: http://www.petefreitag.com/cheatsheets/coldfusion/cfscript/

QueryExecute

- CFQUERY has been the staple to query a database since day 1
- There have been a few attempts to query the database using script
- Here is the latest alternative for script based syntax

Syntax:

QueryExecute(sql_stmt, queryParams, queryOptions);

QueryExecute

Old Script Syntax

```
<cfscript>
   q = new com.adobe.coldfusion.query();
   q.setDatasource("cfartgallery");

    q.setSQL("select * from art
                  where artname like :search or description like :search");

   q.addParam(name="search",value="%e%",cfsqltype="cf_sql_varchar");

   r = q.execute();

</cfscript>
```

QueryExecute

New Script Syntax
queryExecute(sql_stmt, queryParam, queryOptions);

```
Numbers =
(1) queryExecute(“
        SELECT     *
        FROM       art
        WHERE      artname like ‘%this%’
        ORDER BY  id“,
    (2)  {type={value=“number”, cfsqltype=“cf_sql_varchar”}},
          or    {value=“color”},
     (3) {datasource = “scratch_mssql”}
);
```

QueryExecute

queryExecute(sql_stmt, queryParam, queryOptions);

Tag Syntax

```
<cfquery name="qryResult">
        SELECT *
        FROM Employees
</cfquery>
```

Script Syntax

```
qryResult = queryExecute(
  "SELECT * FROM Employees"
);
```

Note: queryExecute is just a function so can be used in tags as well:

```
<cfset temp = queryExecute(…) >
```

QueryExecute

queryExecute(sql_stmt, queryParam, queryOptions);

Tag Syntax

```
<cfset country = “USA” />
<cfset empid = 1 />
<cfquery name=“qryResult”>
    SELECT *
    FROM Employees
    WHERE empid = #empID#
        AND country = #country#
</cfquery>
```

** Should use cfqueryparm

Script Syntax (Parameters using Struct)

```
qryResult = queryExecute(“
    SELECT *
    FROM Employees
    WHERE empid = :empid
        AND country = :country",
  {country="USA", empid=1}
);
```

QueryExecute

queryExecute(sql_stmt, queryParam, queryOptions);

Tag Syntax (Specify a datasource)

```
<cfquery name="qryResult"
        datasource = "myDatasourceName">
    SELECT *
    FROM Employees
</cfquery>
```

Script Syntax (Specify a datasource)

```
qryResult = queryExecute("
    SELECT *
    FROM Employees",
  {},
  {datasource=
          "myDataSourceName"}
);
```

QueryExecute

queryExecute(sql_stmt, queryParam, queryOptions);

Tag Syntax (query with a subquery)

```
<cffunction name="getEqMfg" returntype="query" access="public">
    <cfargument name="dc" cfsqltype="string" required="true">
    <cfquery name="ml" datasource="myDSN">
        Select name, mfgCode, country
        From mfgdetails_tbl
       Where mfgcode in (
          Select mfgCode
          From mfg2devices_tbl
         Where deviceCode =
         <cfqueryparam  cfsqltype="cf_sql_varchar" value="#arguments.dc#" maxlength="5">  )
      Order By country, name
   </cfquery>
  <cfreturn ml >
</cffunction>
```

Script Syntax (query with a subquery)

```
public query function getEqMfg(required string dc){
    var ml = queryExecute("Select name, mfgCode, country
        From mfgdetails_tbl
       Where mfgcode in (
          Select mfgCode
          From mfg2devices_tbl
         Where deviceCode = :dcp
      )
  Order By country, name",
  {dcp={value="dc",cfsqltype="CF_SQL_VARCHAR", maxlength=5 }},
  {datasource = "myDSN"}
);
return ml;
}
```

QueryExecute

queryExecute(sql_stmt, queryParam, queryOptions);

Tag Syntax

(Return a number/count or maxRows)

```
<cfquery name="dsc">
    Select count(*) as qty
    From generic_view"
</cfquery>
<cfquery name="sc" maxrows="#rc#">
   Select specialtyname, qty
   From generic_view
   Order by qty DESC, specialtyname
</cfquey>
```

Script Syntax

(Return a number/count or maxRows)

```
public numeric function getSpecialtyCount(){
    var dsc = queryExecute("Select Count(*) as qty
                From specialtycategorycount_view");
    return dsc.qty;
}
query function getSpecialtyCodeChartData(required numeric rc=99){
    var sc = queryExecute("Select specialtyname, qty
                From specialtycategorycount_view
               Order by qty DESC, specialtyname",
    {},  {maxrows = "rc"});
    return sc;
}
```

QueryExecute

queryExecute: https://helpx.adobe.com/coldfusion/cfml-reference/coldfusion-functions/functions-m-r/queryexecute.html

CFDocs.org: http://cfdocs.org/queryexecute

Query options: http://stackoverflow.com/questions/26296467/what-are-the-options-for-queryoptions-in-the-queryexecute-function

Get rid of cfqueryparam: http://blog.adamcameron.me/2014/03/querycfc-queryexecute-have-good-feature.html

Check out cfuser examples: http://www.cfuser.com/more-queryexecute-examples/

Cfuser: http://www.cfuser.com/more-queryexecute-examples/

Member Functions

- Member Functions are operators and functions that are declared as a member of a class.
- These are more in line with true object oriented style of coding
- Member functions can also be chained together

Formats:

Old:

```
ArrayAppend(empArr, emp)
structIsEmpty(empStruct)
```

New:

```
empArr.append(emp)
empStruct.isEmpty()
```

Member Functions

List of data types supported:

- String: stringVar.find()
- List: listVar.listFind()
- Array: arrayVar.find()
- Struct: structVar.find()
- Date: dateVar.dateFormat()
- Image: someImage.crop()
- Query: queryVar.addColumn()
- Spreadsheet: spreadsheetVar.writ()
- XML: xmlVar.search()

Chaining: stringVar.find(“I like cfml”, “cfml”).uCase()

Member Functions

Regular CF function syntax

```
dateTimeFormat(
        dateAdd( 'd', -5, now() )
        , 'yyyy-mm-dd' );
```

```
dateTimeFormat( dateAdd( 'd', -5, now() ), 'yyyy-mm-dd' );
```

Note: dateTimeFormat function was added in CF10, combo of dateTime and timeFormat.

Member Function syntax

```
now().add( 'd', -5 )
.dateTimeFormat( 'yyyy-mm-dd' );
```

```
now().add( 'd', -5 ).dateTimeFormat( 'yyyy-mm-dd' );
```

Member Functions

A chaining example:

```
s =    "The";
s =    s.append("quick brown fox", " ")
    .append("jumps over the lazy dog", " ")
        .ucase()
            .reverse();
writeOutput(s);
```

Result: GOD YZAL EHT REVO SPMUJ XOF NWORB KCIUQ EHT

s.append("quick brown fox", " ").append("jumps over the lazy dog", " ").ucase().reverse();

Member Functions

Regular CF function syntax

```
var myArray = ArrayNew(1);
    ArrayAppend(myArray, "objec_new");
    ArraySort(myArray, "ASC");
```

Member Function syntax

```
myArray.append("objec_new");
myArray.sort("ASC");
```

Member Functions

- Important Note on a potential Member Function Gotcha:
  - Some member functions might fall into underlying Java methods if the strict ColdFusion syntax is not followed.
  - https://bugbase.adobe.com/index.cfm?event=bug&id=3753710

Member Functions

- Resources

Adobe ColdFusion wiki: https://helpx.adobe.com/coldfusion/developing-applications/building-blocks-of-coldfusion-applications/using-the-member-functions.html

*** You need to go to this like to see all the new supported member functions for ColdFusion 2016 ***

Elvis Operator

- The Elvis Operator added in ColdFusion 11
- It works like a Ternary Operator; it's a decision making operator that requires three operands: condition, true statement, and false statement that are combined using a question mark (?) and a colon (:):
  - ((condition) ? trueStatement : falseStatement)
- The way it works is that the condition is evaluated. If it is true, then the true statement executed; if it is false, then the false statement executes
- Before Elvis we had isDefined(), structKeyExists() and IF statements to do these kind of evaluations.

Elvis Operator

- Syntax: ?:

```
value = ( url.foo ?: "bar" );
```

- The Elvis operator is primarily used to assign the ‘right default’ for a variable or an expression
- Or it is a short-hand way to do parameterization. It will allow us to set a value if the variable is Null

Elvis Operator

Examples which are all the same:

1. If (isNull(local.testVar)){
        value = “test Item”;
   } else{
        value = local.newTest;
   }

2. Value = (local.testVar **?:** “test Item”);

3. Value = (isNull(local.testVar) **?** “test Item”**:** local.newTest);

Elvis Operator

```
result = firstOperand ?: secondOperand;       // binary
result= (local.myVar ?: “default value”);
```

OR

```
result = firstOperand ?: secondOperand;       // binary
result = isInteger(17) ? "it's an integer" : "no it isn't";       // "it's an integer"
```

OR

```
result = firstOperand ? secondOperand : thirdOperand;     // ternary
result = isInteger("nineteen") ? "it's an integer" : "no it isn't";      // "no it isn't“
```

Elvis Operator

- Resources:
- Wiki docs https://helpx.adobe.com/coldfusion/developing-applications/the-cfml-programming-language/elements-of-cfml/elvis-operator.html

Closures

- A Closure is a function which binds variable references at declaration time not at use-time
- Callbacks are not Closures (inline callbacks are)

- That inner function has access to the var scope of the function it was defined from. This is what a **closure** is. It knows about its origin and it doesn't forget. It will always be tied to that same parent var scope.

Closures

javascript example (works in CF also)

```
Function outerFunction() {
        var a = 3;
        return function innerFunction(b){
        var c = a + b;
        return c;
        }
}
(1) var foo = outerFunction()
(2) var result = foo(2);
(3) Console.log(result); //5
```

- We have an **outer function** with a nested function which accepts a parameter b
- (1)When you invoke the outer you get the **inner returned** later.
- (2)Notice the outer function was called but the a still has its value and is used in the return function (innerFunction).
- (3)That is why the result Is 5!
- http://taha-sh.com/blog/understanding-closures-in-javascript

Closures

```
function bunchofstuff(…) {    // OLD CODE
            …bunch of stuff…
}
function restofstuff(…) {     …rest of stuff… }

function stuff1(…) {   bunchofstuff(…);   …more stuff… restofstuff(…);   }
function stuff2(…) {  bunchofstuff(…);    …other stuff…  restofstuff(…);  }
```

You still have 4 functions and stuff1 and 2

still have common code

```
function generalstuff(…, differentstuff) {    // CLOSURE WAY
   …bunch of stuff…    differentstuff(…);   …general code…
}
function stuff1(…) {
     generalstuff(…, function(…){ …more specific stuff…
   });
}
function stuff2(…) {
    generalstuff(…, function(…){  …more specific stuff…     });
}
```

Using the closure code above all common code is in general stuff and…

you call stuff1 or stuff 2 for the specific code you need run (Template Method design pattern)

Closures

ColdFusion Docs Example:

```
function helloTranslator(String helloWord) {
    return function(String name) {
        return "#helloWord#, #name#";
    };
}
helloInHindi=helloTranslator("Namaste");
helloInFrench=helloTranslator("Bonjour");

writeoutput(helloInHindi("Anna"));
```

→ Namaste, Anna

```
writeoutput(helloInFrench("John"));
```

→ Bonjour, John

- In this case, using closure, two new functions are created. One adds Namaste to the name. And the second one adds Bonjour to the name.
- helloInHindi and helloInFrench are closures. They have the same function body; however, store different environments.

- The inner function is available for execution after the outer function is returned. The outer function returns the closure
- A closure is formed when the inner function is available for execution. Meaning, it is formed when it is defined, not when it is used.

Closures

- Ray Camden example (technically is, but more like a callback/inline function) :

```
Data = [“Neil Diamond”, “Depeche Mode”, “The Cure”, “Cher”, “Ace of Base”, “Frank Sinatra”, The Church”];
arraySort(data, function(a,b){
        var first = a;
        var second = b;
        first = replace(first, “The “,  “”);
        second = replace(second, “The “, “”);
    return first gt second;
});
```

This code above will order the data array by names excluding THE.

Ace of Base, Cher, The Church, The Cure, Depeche Mode, Frank Sinatra, Neil Diamond

Closures

ColdFusion built in Functions that use Closures:

- CF10 Closure Functions:
  - ArrayEach, StructEach
  - ArrayFilter, StructFilter,ListFilter
  - ArrayFindAt, ArrayFindAllNoCase
- CF11 Closure Functions:
  - isClosure
  - ArrayReduce, StructReduce, ListReduce
  - ArrayMap, StructMap, ListMap
- CF 2016 added each, map, reduce & filter for Queries

Closures

ArrayEach

```
arrayEach(
    array,
    function(any currentObj)
     {
      }
);
```

```
arrayEach([1,2,3,4,5,6,7], function(item)
{
    writeOutput
     (dayOfWeekAsString(item) & “<br>”);
});
```

Answer:

Sunday

Monday

Tuesday

Wenesday

Thursday

Friday

Saturday

Closures

structEach

```
structEach(
    struct,
    function(key, value)
    {
     }
);
```

```
structEach({one:1, two:2}, function(key, value)
{
    writeoutput(key & ":" & value);
});
```

Answer:

ONE: 1

TWO: 2

Closures

```
arrayFilter(
    array,function(arrayElement,
    [,index, array])
     {return true|false;}
);
```

Note: CF 11 - the callback functions of all the filters have access to the entire array/list/struct apart from the current element

- ArrayFilter

```
Filtered =
    arrayFilter([“myNewBike”,
         “oldBike”], function(item)
{
     return len(item) > 8;
});
```

Answer

=> [1]

Closures

```
arrayFindAll
ArrayFindAll(array, object)   or
ArrayFindAll(
  array,
  function(arrayElement) {
    return true|false;
}
)
```

```
Data=[{age:4}, {age:6}, {age:41}];
Answer =
arrayFindAll(data,
                function(item){
    return item.age < 10;
});
Answer
[1,2]
```

Closures

arrayReduce

ArrayReduce(

array,

function(result, item, [,index, array])

[, initialValue]

)

```
complexData = [{a:4}, {a:18}, {a:51}];
Function reducer(prev, element){
    return prev + element.a;
}
sumOfData =
 arrayReduce(complexData, reducer, 0);
```

Answer

73

Closures

arrayMap

ArrayMap(
  array,
  function(item [,index, array])
)

```
complexData = [{a:4}, {a:18}, {a:51}];
newArray =
arrayMap(complexData, function(item){
    return item.a;
}, 0);
```

Answer

[4, 18, 51]

Closures

Testbox: https://www.ortussolutions.com/product/testbox

- TestBox uses Closures
- Since the implementations of the describe() and it() functions are closures, they can contain executable code that is necessary to implement the test. All ColdFusion rules of scoping apply to closures, so please remember them. We recommend always using the variables scope for easy access and distinction.
- A test suite begins with a call to our TestBox **describe()** function with at least two arguments: a *title* and a **closure** within the life-cycle method called **run()**. The title is the name of the suite to register and the function is the block of code that implements the suite.

```
function run() {
  describe("A suite is a closure",
  function() {
    c = new Calculator();
      it("and so is a spec", function() {
        expect( c ).toBeTypeOf( 'component' );
      });
  });
}
```

Closures

- Some Example code online or resources:
  - Sesame library inline functions/closures
    https://github.com/markmandel/Sesame
  - Underscore.cfc ported from underscore.js
    https://github.com/russplaysguitar/UnderscoreCF
  - TestBox gives the ability to use closures
    http://wiki.coldbox.org/wiki/TestBox.cfm
  - Adam Tuttle presentation on closures
    http://fusiongrokker.com/p/closures/#/
  - Adam Cameron blog multiple Closure examples
    http://blog.adamcameron.me/search/label/Closure

Map and Reduce (and more)

- The **map()** functions iterate over the collection (be it a list, array or struct), and returns a new object with an element for each of the ones in the original collection. The callback returns the new element for the new collection, which is derived from the original collection. It is important to note that only the collections values are changed, it will still have the same key/indexes.
- So it remaps the original collection
  - ArrayMap, StructMap, ListMap, (query.map)
- The **reduce()** operation is slightly more complex. Basically it iterates over the collection and from each element of the collection, derives one single value as a result.
  - ArrayReduce, StructReduce, ListReduce, (query.reduce)

Map and Reduce (and more)

- The **Filter()** function is similar to map and reduce. It will iterate over a given object and return a new object but the original list is unaffected. The callback returns a Boolean to determine if the element is returned in a new object.

```
structEach(circles,function(key,value)
{
   matchednames =
    arrayfilter(value,function(obj)
    {
      return left(obj,1)=='d';
    }
});
```

Example shows how to look at a struct and if it begins with a D, return it.

Map and Reduce (and more)

- The **each()** function iterates over a given object and calls a callback for each element. This can be used instead of a loop.

```
letters = ["a","b","c","d"];
arrayEach(letters, function()
{
   writeDump(arguments);
});
Struct 1 a
struct 1 b
struct 1 c
struct 1 d
```

Map and Reduce (and more)

- Map()

```
complexData = [ {a: 4}, {a: 18}, {a: 51} ];
newArray =
arrayMap( complexData, function(item)
{
    return item.a;
}, 0 );
Answer: [4, 18, 51]
```

Map and Reduce (and more)

- Reduce()

```
complexData = [ {a: 4}, {a: 18}, {a: 51} ];

sum =

arrayReduce( complexData, function(prev, element)

{

    return prev + element.a;

}, 0 );
```

Answer: 73

Map and Reduce (and more)

- ArrayMap Example (next few examples from Adam Cameron):

```
dcSports    = ["Redskins", "Capitals", "Nationals", "Wizards", "United"];
colourInList = arrayMap(
      dcSports, function(v,i,a){
            return replace(a.toList(), v, ucase(v) );
       }
);
writeDump([dcSports,colourInList]);
```

Will see an array of dcSports with 7 items each showing the next in the row all caps:

REDSKINS,Capitals,Nationals,Wizards,United

Redskins,**CAPITALS**,Nationals,Wizards,United

Redskins,Capitals,**NATIONALS**,Wizards,United

Map and Reduce (and more)

- StructMap()

```
original = {"one"={1="redskins"},"two"={2="capitals"},"three"={3="nationals"},"four"={4="wizards"}};
fixed = structMap(original, function(k,v)
{
   return v[v.keyList()];
});
writeDump([ original, fixed]);
```

This just returns the **digit as a key** and **sports team as a value**

1 wizards

2 redskins

3 nationals

4 capitals

Map and Reduce (and more)

- listMap()

```
dcSports = "Redskins,Capitals,Nationals,Wizards,United";
externalList = "";
reverseSports = listMap( dcSports, function(v,i,l)
{
      var newValue = "#i#:#v.reverse()#";
      externalList = externalList.append(newValue);
   return newValue;
});
writeDump([{dcSports=dcSports},{reverseSports=reverseSports},{externalList=externalList}]);
```

This should reverse the dcSports list and show it backwards.

Map and Reduce (and more)

```
colours = queryNew("id,en,mi", "integer,varchar,varchar", [
    [1,"red","whero"],
    [2,"orange","karaka"],
    [3,"yellow","kowhai"],
    [4,"green","kakariki"],
    [5,"blue","kikorangi"],
    [6,"indigo","poropango"],
    [10,"violet","papura"]
]);

maoriColours = colours.map(function(colour, index, colours){
    return {mi=colour.mi};
}, queryNew("mi","varchar"));

writeDump(var=maoriColours);
```

query	
	MI
1	whero
2	karaka
3	kowhai
4	kakariki
5	kikorangi
6	poropango
7	papura

Notice that map takes a second argument which Is a second query. This acts as a template for the Remapped query.

Map and Reduce (and more)

- listReduce()

```
numbers = "1,2,3,4,5,6,7,8,9,10";
sum = listReduce(numbers, function(previousValue, value)
{
	return previousValue + value;
},  0);
writeOutput("The sum of the digits #numbers# is #sum#<br>");
```

The sum of the digits 1,2,3,4,5,6,7,8,9,10 is 55

Map and Reduce (and more)

- arrayReduce()

```
dcSports = ["Redskins", "Capitals", "Nationals", "Wizards", "United"];
ul = arrayReduce(dcSports, function(previousValue, value)
  {
      return previousValue & "<li>#value#</li>";
   },   "<ul>"
) & "</ul>";
writeOutput(ul);
```

Result is just the **Array turned into a list**

Map and Reduce (and more)

- structReduce()
  - StructReduce(struct, function(result, key, value [,struct]), initialVal)

```
rainbow = { "Red"= "Rojo",  "Orange"= "Anaranjado", "Yellow"= "Amarillo",  "Green"= "Verde" };
 dl = structReduce(  rainbow, function(previousValue, key, value)
   {
      return previousValue & "<dt>#key#</dt><dd>#value#</dd>";
    },
    "<dl>"
) & "</dl>";
writeOutput(dl);
```

Result is a **two column list** of the key and the value: Red Rojo

Map and Reduce (and more)

```
week = queryNew("id,en,mi", "integer,varchar,varchar", [
   [1,"Monday", "lunes"],
   [2,"Tuesday", "martes"],
   [3,"Wednesday", "miercoles"],
   [4,"Thursday", "jueves"],
   [5,"Friday", "vernes"],
   [6,"Saturday", "sabado"],
   [7,"Sunday", "domingo"]
]);

shortestMaoriDayName = week.reduce(function(shortest,number){
   if (shortest.len() == 0) return number.mi;
   return number.mi.len() < shortest.len() ? number.mi : shortest;
}, "");

writeOutput(shortestMaoriDayName);
```

Result (which spanish name is the smallest using elvis operator):

lunes

Map and Reduce (and more)

- Resources
  - map and reduce & more with adam: http://blog.adamcameron.me/2014/02/coldfusion-11-map-and-reduce.html
  - map reduce: https://hacks.mozilla.org/2015/01/from-mapreduce-to-javascript-functional-programming/
  - js working example of each: http://elijahmanor.com/reducing-filter-and-map-down-to-reduce/
  - ColdFusion 2016 query iterations: http://blog.adamcameron.me/2016/02/coldfusion-2016-query-iteration.html

First Class Functions

- A first class object is the one which could be passed as an argument
- Now you can treat ColdFusion functions such as arrayLen, lCase, uCase, etc. as objects and you can do the following:
  - Pass them as arguments to a function call
  - Assign them to variables
  - Return them as a result of a function invocation
- This first introduced in ColdFusion 11

First Class Functions

- What is a Callback function?
  - This is a function that is passed into another function.
  - You build a main generic function to do work, then you can pass in a more detailed function to do a lot of the work.
    - Sorting, you might want to sort money data, or state data, or city data
      - Build a sort function then pass in the callback function you need to do your sort

```
Sort(dataToSort, callBackFunction)
    {  //do stuff }

callbackFunction()
   { //build sort }
```

Calls the Sort function:

```
sortedData = sort(unsortedData, callbackFunction);
```

First Class Functions

Here is how to pass in a built-in function as an argument:

```
Function convertCaseArray(Array, array, function converter){
        for (var i=1, I <= arrayLen(array); i++){
                array[i] = converter(array[i]);
        }
        return array;
}
resultArray = convertCaseArray(['One','Two','Three'], lcase);
Writedump(resultArray);
```

Array converted to all lower case

First Class Functions

This is where lcase and ucase are being returned from a function:

```
Function convertArray(array array, string caseTo){
        caseConvertFunction = getConvertFunction(caseTo);
        for (var i=1, I <= arrayLen(array); i++){
                array[i] = caseConverterFunction{array[i]);
        }}
Function getConvertFunction (String caseType){
        if (caseType == 'lower') return lcase; else return ucase;
}
resultArray_lower = convertArray(['One', 'Two', 'Three'], 'lower');
resultArray_upper = convertArray(['One', 'Two', 'Three'], 'upper');
```

_lower array: one, two, three

_upper array: ONE, TWO, THREE

First Class Functions

- Resources
  - http://blogs.coldfusion.com/post.cfm/language-enhancements-in-coldfusion-splendor-promoing-built-in-cf-function-to-first-class
  - http://blog.adamcameron.me/2013/09/wow-coldfusions-built-in-functions-will.html
  - http://blog.adamcameron.me/2012/09/callbacks-function-expressions.html
  - Sesame project: https://github.com/markmandel/Sesame
  - Adobe Online Docs: https://helpx.adobe.com/coldfusion/developing-applications/the-cfml-programming-language/built-in-functions-as-first-class-citizen.html

ColdFusion 12 feature

- http://www.adobe.com/devnet/coldfusion/articles/whats-new-cf-2016.html
- Language Enhancement

Save Navigation

- Save Navigation -- **?.**
  - Save navigation operator can be used to access members of a struct or values of an object.
  - Normally we use a .
    - <cfset t = myObject.getUserObj />
  - Now we can use the ?. To ensure that an exception is not thrown when the variable is not defined
    - <cfset t = myObject?.getUserObj />
    - If myObject is not defined then t will be assigned as undefined

Save Navigation

Old way:

```
if(isDefined("foo"))
 writeOutput(foo.bar());
else
 writeOutput("not defined");
```

New way:

```
writeOutput(foo?.bar());
```

Chaining:

```
writeOutput(employee?.name?.firstname?.trim());
```

The above code does not output any value if any of the following is undefined/null:

- employee OR
- employee.name OR
- employee.name.firstname.

Note: the code will throw an exception if the employee is defined but employee.name is not

Ordered collection

StructNew("Ordered") creates a struct that maintains insertion order.

Currently, keys are fetched in random order while looping over the struct.

Loop over the struct using StructNew(“Ordered”), keys are fetched according to the insertion order.

Ordered collection

```
<!--- Create a structure and set its contents. --->
<cfset departments=StructNew("Ordered")>
<cfset val=StructInsert(departments, "John", "Sales")>
<cfset val=StructInsert(departments, "Tom", "Finance")>
<cfset val=StructInsert(departments, "Mike", "Education")>
<cfoutput>
<cfloop collection=#departments# item="person">
#person# ---  #Departments[person]#<br>
 </cfloop>
</cfoutput>
```

Output:

John Sales

Tom Finance

Mike Education

PassArrayByReference

Arrays -- Pass by reference

By default, arrays are passed by value, which is slower as compared to passing arrays by reference. You can now choose to pass arrays by reference by using a new application setting called passArrayByReference. It accepts a boolean value.

When set to true, if an array is passed as a UDF argument, it is passed by reference.

In Application.cfc file

PassbyReference = ‘True’;

SearchImplicitScopes setting

Generally, a referenced variable is searched in various scopes before a match is found. In cases where variable is not found in common scopes like function, local scope, and variable scopes, the search continues in implicit scopes, which can be time consuming. Implicit scope search can be disabled by using a new application setting called searchImplicitScopes. It accepts a boolean value.

When set to false, an un-scoped variable is not searched in implicit scopes.

In Application.cfc file

searchImplicitScopes = ‘True’;

Command Line Interface (CLI)

  - Operations – File, Database, Network, Webservice
  - Relative or Absolute Path
  - Arguments and Named arguments – can be read back with a number of methods
  - Read(), writeLn(), writeError()
- Unsupported features listed in documentation
  - Arg methods: cli.getArgs(), getNamedArgs(), getNamedArgs(), getUnnamedArgs()
    Read(), writeLn(), writeError() - to read and write to stdin and stdout/stderr

    Unsupported: pdf/document, chart, websocket, etc.
- https://helpx.adobe.com/coldfusion/2016/command-line-interface.html
- http://blog.adamcameron.me/2015/11/coldfusion-12-long-awaited-cli.html

ColdFusion 12 Feature

- There are a few other new Language enhancements not discussed today
- There are new enhancements to current features of ColdFusion

Resources:

https://helpx.adobe.com/coldfusion/whats-new.html

http://www.adobe.com/products/coldfusion-enterprise/features.html

Final Thoughts/Resources

- These topics were only some more modern concepts for CFML.
- Where to get more information from CFML Community
  - Adobe Wiki
  - CFDocs.org
  - Cfblogger (put in aggregator like feedly)
  - Slack Channel – 1200+ cfml users https://cfml.slack.com/
  - Trycf.com (place to run examples)
  - Twitter, Facebook, youTube, Vimeo and other social sites

Dan Fredericks

@fmdano74

fmdano@gmail.com

I am on most social networks, twitter, facebook, G+, LinkedIn

Special Thanks to Adam Cameron, Adam Tuttle, Ray Camden, Sean Corfield, and Brad Wood for examples