3

Arrays

3.0. Introduction

An array is a data type containing an ordered collection of variables. Items in an array are referred to as *array elements* and are referenced by their *index* (or numerical position) in the array.

You use arrays when you have a collection of variables and need to store them in a defined order. Like lists (discussed in Chapter 2, "Working with Lists"), arrays support variable order—you can explicitly reference the value stored in a given position of the array. Although lists only store simple variables like strings and numbers, arrays can store complex data types, such as structures, queries, and even other arrays. If you need to store complex data types in a collection but do not need them in a defined order, you may want to look into using the ColdFusion data type structure, discussed in Chapter 4, "Structures."

This chapter shows you how to iterate over array element values, manipulate element values, and sort an array. It also explains how and when to use multidimensional arrays and how and when to use array aggregate functions.

3.1. Creating an Array

You want to create a new array.

Technique

Use the `ArrayNew()` function to create a new ColdFusion `Array` object.

```
<cfset aPetSounds = ArrayNew(1)>
```

Comments

ColdFusion arrays are different than arrays in many other programming languages in that ColdFusion arrays are sized *dynamically*. A dynamically sized array is one that allows for expansion as elements are added—the programmer does not need know the size of the array when it is declared. Instead, ColdFusion sizes the array as the number of elements exceeds the size that ColdFusion initially appropriated.

This dynamic sizing provides flexibility in programming but can also cause a performance hit—especially for large arrays, because ColdFusion essentially has to guess how large the array will eventually be. So if you know the maximum number of elements your array will contain, ColdFusion provides the `ArrayResize()` function to force a minimum size. `ArrayResize()` accepts two parameters, the array to resize and the minimum size to set that array. ColdFusion documentation suggests using the `ArrayResize()` function immediately following an array declaration that will contain more than 500 elements.

Multidimensional arrays are covered later in this chapter—until then you'll use only single dimensional arrays, created by passing a value of `1` to the `ArrayNew()` function.

Note
As a best practice, all arrays in this chapter begin with the letter *a*, as in `aArrayName`.

3.2. Adding an Element to an Array

You want to add an element to an array.

Technique

You add elements to an array by statically referencing the index of the element you want to insert, by using the `ArrayAppend()` function, or by using the `ArrayPrepend()` function.

```
<!---add element by index--->
<cfset aPetSounds[1] = "You Still Believe In Me">
<cfset aPetSounds[2] = "That's Not Me">

<!---add element using ArrayAppend--->
<cfset ArrayAppend(aPetSounds, "Don't Talk (Put Your Head On My Shoulder)")>

<!---add element using ArrayPrepend--->
<cfset ArrayPrepend(aPetSounds, "Wouldn't It Be Nice")>
```

Comments

After you've created an array, the natural next step is to populate its elements. (To create an array, see section 3.1, "Creating an Array.") One way to set the index value of a given element in an array is to reference the element you want to populate by index and set its value explicitly. This code is an example of creating a new array named `aPetSounds` and setting the first and second element to the strings `"You Still Believe In Me"` and `"That's Not Me"`, respectively, by reference. (*Pet Sounds*, of course, is an album by the Beach Boys—often considered their masterpiece.)

Notice that ColdFusion uses 1-based numbering arrays—the first array element is in position 1. Many programming languages use 0-based numbering arrays, which place the first element in position 0.

Another way to add an element to an array is to use either the `ArrayAppend()` or `ArrayPrepend()` function. This method of adding elements ensures you don't place a value in an unintended index of the array.

`ArrayPrepend()` will prepend your value to the array, making it the first element and pushing all other elements down an index. This is a useful function if you need to add an element to the beginning of an array, but aren't worried about the other elements' index values. For example, in the previous technique, `Wouldn't It Be Nice` gains an index of `1`, whereas `You Still Believe In Me` and `That's Not Me` now have index values of `2` and `3`, respectively (even though their index values were given explicitly).

`ArrayAppend()` adds your element to the end of an array, making it the last element in the array, and leaving the other elements where they are. Use the `ArrayAppend()` function when you need to maintain the integrity of the array's current index. Notice that `Don't Talk (Put Your Head On My Shoulder)` is now a part of the `petSounds` array, but it has simply been "tacked on," and has an index value of `4`. This leaves the first three song titles' index values untouched.

3.3. Displaying a Value in an Array

You want to retrieve and display a value stored in an array.

Technique

Use `<cfoutput>` tags and reference the desired value by index to retrieve and display the value stored in a specific array element.

```
<cfoutput>
   Song 1: #aPetSounds[1]#<br>
   Song 2: #aPetSounds[2]#<br>
```

```
   Song 3: #aPetSounds[3]#<br>
   Song 4: #aPetSounds[4]#
</cfoutput>
```

Comments

Executing this code produces the following output:

```
Song 1: Wouldn't It Be Nice
Song 2: You Still Believe In Me
Song 3: That's Not Me
Song 4: Don't Talk (Put Your Head On My Shoulder)
```

If you attempt to reference an element that does not exist in your array, ColdFusion throws an `ArrayBoundException` exception. Refer to Chapter 8, "Exception Handling," for more information on exceptions in CFMX.

3.4. Looping over an Array

You want to iterate over each element in an array.

Technique

Use `<cfloop>` to iterate over each element in an array.

```
Pet Sounds Track Listing
<hr>
<cfoutput>
   <!---loop over array and display each element value--->
   <cfloop from="1" to="#ArrayLen(aPetSounds)#" index="i">
      Song #i#: #aPetSounds[i]#<br>
   </cfloop>
</cfoutput>
```

Comments

This example revisits the `aPetSounds` array and loops from `1` to the length of the array to extract and display the value of each element. Looping over an array to display its values ensures that you won't reference an index that doesn't exist. This code also shows that you can use ColdFusion expressions and variables to reference indexes within an array—in this case, you use the current index of the loop to retrieve the associated index in the array.

> **Note**
> This code introduces the `ArrayLen()` function. `ArrayLen()` accepts a single argument: the array in question. Similar to the `Len()` function, which counts the number of characters in a string, `ArrayLen()` returns the number of elements in an array.

3.5. Manipulating Array Values

You want to access or modify specific elements within an array.

Technique

Manipulate elements in a populated array by iterating over the array and performing the desired function during each iteration of the loop.

```
<!---Manipulate each element to append song number--->
<cfloop from="1" to="#ArrayLen(aPetSounds)#" index="i">
   <cfset aPetSounds[i] = "Song " & i & ": " & aPetSounds[i]>
</cfloop>
```

Comments

Arrays are used in Web applications to store ordered collections of related data. Naturally, as a user progresses through an application, data will change. Therefore, you need a way to manipulate the values of an array in order to perform functions on element values. Say you need to add the track number to each value in the `aPetSounds` array. You can do so by looping over the array values and manipulating the string to include the loop number, resulting in the desired value.

Notice in the first `cfloop` statement that the value of the current element is referenced and a new value is set in the same line. This inline variable manipulation is acceptable in CFML. You also use the index variable *`i`* to insert the song number—a simple example of benefiting from the order functionality provided by the array data type. To display the new values with the track number prepended to the value, execute the following code:

```
Pet Sounds Track Listing
<hr>
<cfoutput>
   <!---loop over array and display each element value--->
   <cfloop from="1" to="#ArrayLen(aPetSounds)#" index="i">
      #aPetSounds[i]#<br>
   </cfloop>
</cfoutput>
```

3.6. Sorting an Array

You want to sort the elements of an array.

Technique

Use the `ArraySort()` function to sort the elements of an array.

```
<cfset abcList = "c,B,b,A,C,a">
```

```
<cfset aAbcArray = ArrayNew(1)>
<cfloop from="1" to="#ListLen(abcList)#" index="i">
   <cfset aAbcArray[i] = listGetAt(abcList,i)>
</cfloop>
aAbcArray:<br/>
<cfoutput>
   <cfloop from="1" to="#ArrayLen(aAbcArray)#" index="i">
      element #i#: # aAbcArray[i]#<br/>
   </cfloop>
</cfoutput>
<p/>
aAbcArray sorted with "text" attribute:<br/>
<cfset ArraySort(aAbcArray,"text")>
<cfoutput>
   <cfloop from="1" to="#ArrayLen(aAbcArray)#" index="i">
      element #i#: # aAbcArray[i]#<br/>
   </cfloop>
</cfoutput>
<p/>
aAbcArray sorted with "text" and "desc" attributes:<br/>
<cfset ArraySort(aAbcArray,"text","desc")>
<cfoutput>
   <cfloop from="1" to="#ArrayLen(aAbcArray)#" index="i">
      element #i#: # aAbcArray[i]#<br/>
   </cfloop>
</cfoutput>
<p/>
aAbcArray sorted with "textnocase" attribute:<br/>
<cfset ArraySort(aAbcArray,"textnocase")>
<cfoutput>
   <cfloop from="1" to="#ArrayLen(aAbcArray)#" index="i">
      element #i#: # aAbcArray[i]#<br/>
   </cfloop>
</cfoutput>
<p/>
aAbcArray sorted with "textnocase" and "desc" attributes:<br/>
<cfset ArraySort(aAbcArray,"textnocase","desc")>
<cfoutput>
   <cfloop from="1" to="#ArrayLen(aAbcArray)#" index="i">
      element #i#: # aAbcArray[i]#<br/>
   </cfloop>
</cfoutput>
```

Comments

The `ArraySort()` function sorts arrays numerically or alphabetically. It accepts two required arguments, the array to be sorted and the sort type, and one optional argument, sort order (which, if not provided, defaults to ascending).

Using `ArraySort()`, the sort type argument can be `numeric`, `text`, or `textnocase`. If sort type is `numeric`, the array elements are sorted numerically in ascending or descending order based on the sort order argument. If no sort order is provided, `ArraySort()` uses ascending, the default.

If sort type is `text`, the array elements are sorted alphabetically taking case into account. In text sorts, sort order ascending starts with uppercase *A* and ends with lowercase *z*, descending starts with lowercase *z* and ends with uppercase *A*. If sort type is `textnocase`, the array elements are sorted alphabetically without taking case into account. That is, if sort order is ascending, *A/a* precedes a *B/b*. If sort order is descending, a *Z/z* precedes a *Y/y*. Take a look at what happens when you execute the code in this technique:

```
aAbcArray:
element 1: c
element 2: B
element 3: b
element 4: A
element 5: C
element 6: a

aAbcArray sorted with "text" attribute:
element 1: A
element 2: B
element 3: C
element 4: a
element 5: b
element 6: c

aAbcArray sorted with "text" and "desc" attributes:
element 1: c
element 2: b
element 3: a
element 4: C
element 5: B
element 6: A

aAbcArray sorted with "textnocase" attribute:
element 1: a
element 2: A
element 3: b
```

```
element 4: B
element 5: c
element 6: C

aAbcArray sorted with "textnocase" and "desc" attributes:
element 1: C
element 2: c
element 3: B
element 4: b
element 5: A
element 6: a
```

3.7. Multidimensional Arrays

You want to create an array with more than one dimension.

Technique

Use `ArrayNew()`, passing in as a parameter the number of dimensions you require, to create a multidimensional array.

```
<!---Create a two dimensional aBeachBoysAlbums array--->
<cfset aBeachBoysAlbums = ArrayNew(2)>

<!---Build first album array, Surfin' Safari--->
<cfset aBeachBoysAlbums[1][1] = "Surfin' Safari">
<cfset aBeachBoysAlbums[1][2] = "County Fair">
<cfset aBeachBoysAlbums[1][3] = "Ten Little Indians">

<!---Build second album array, Surfin' USA--->
<cfset aBeachBoysAlbums[2][1] = "Surfin' USA">
<cfset aBeachBoysAlbums[2][2] = "Farmer's Daughter">
<cfset aBeachBoysAlbums[2][3] = "Miserlou">

<!---Build third album array, Surfer Girl--->
<cfset aBeachBoysAlbums[3][1] = "Surfer Girl">
<cfset aBeachBoysAlbums[3][2] = "Catch a Wave">
<cfset aBeachBoysAlbums[3][3] = "The Surfer Moon">
```

Comments

A multidimensional array is an array of arrays rather than an array of single values. Until this point in the chapter, one-dimensional arrays have been covered, and the array elements have all been simple values—the name of a song or a letter of the alphabet. In a multidimensional array, you use the array elements to store other arrays.

ColdFusion allows you to define up to three dimensions in an array with a call to the `ArrayNew()` function. `ArrayNew()` takes one argument—`1`, `2`, or `3`—depending on how many dimensions you need.

This code is an example of creating a two-dimensional array. It continues using the Beach Boys as a model and uses a two-dimensional array to store album information in order of release. For this example, you use only the first three songs from each of their first three releases.

If you want to display the values stored in the inner dimensions of an array, it is necessary first to get a handle on each element. Using `cfloop` is the best way to do so. Remember, you need exactly as many loops as you have dimensions in the array.

3.8. Creating Arrays with More Than Three Dimensions

You want to create an array with more than three dimensions.

Technique

Define multiple dimensions in an array by creating three dimensions at a time with `ArrayNew()`.

```
<!---create three dimensional aDeepArray array--->
<cfset aDeepArray = ArrayNew(3)>

<!---build nine dimensions--->
<cfset aDeepArray[1][1][1] = ArrayNew(3)>
<cfset aDeepArray[1][1][1][1][1][1] = ArrayNew(3)>
<cfset aDeepArray[1][1][1][1][1][1][1][1][1] = "I'm in the ninth dimension">

<!----display first element in ninth dimension--->
<cfoutput>
   #aDeepArray[1][1][1][1][1][1][1][1][1]#
</cfoutput>
```

Comments

Although `ArrayNew()` lets you define only three dimensions, should the need arise you can bypass this constraint by setting any of the three dimension values to an array itself and repeat until you have the dimension depth you require. The code in this example places a string value in the lowly ninth dimension.

3.9. Array Aggregate Functions

You want to perform aggregate functions on array elements.

Technique

Use ColdFusion array aggregate functions to perform a function on all elements within an array.

```
<!---Create aArray array and set array values--->
<cfset numList = "0,5,22,7,31">
<cfset aArray = ListToArray(numList)>
aArray aggregate functions
<hr>
<cfoutput>

<!---Perform aggregate functions on aArray and display results--->

Maximum Value: #ArrayMax(aArray)#<br>

Minimum Value: #ArrayMin(aArray)#<br>

Sum: #ArraySum(aArray)#<br>

Average: #ArrayAvg(aArray)#
</cfoutput>
```

Comments

Aggregate functions are functions executed on multiple values—array aggregate functions are functions executed on each array element value.

Table 3.1 defines the four functions available in ColdFusion MX.

Table 3.1 **CFMX Array Aggregate Functions**

Function	Return Value
ArrayMax(array)	Returns the maximum numeric value in array.
ArrayMin(array)	Returns the smallest numeric value in array.
ArrayAvg(array)	Returns the numeric average of all values in array.
ArraySum(array)	Returns the numeric sum of all values in array.

Two exceptions of note when dealing with array aggregate functions are as follows:

- If there is a non-numeric value in an array passed into an aggregate function, ColdFusion throws a runtime exception.
- If the array passed into an aggregate function is empty, each of the functions returns a `0`.

Note that ColdFusionMX returns `0` when an empty array is passed into an array aggregate function. ColdFusion 5 returned infinity.

3.10. Array Utility Functions

You want to perform utility functions on arrays.

Technique

Use ColdFusion array utility functions.

```
<cfset aNewArray = ArrayNew(3)>
```

Comments

You have used a number of CFMX array utility functions in various techniques throughout this chapter. Table 3.2 defines each of the array utility functions, including ones you haven't yet used. Don't worry if you don't understand them all yet—the more you work with arrays, the more you'll find the need to reference this table.

Table 3.2 **ColdFusion Array Utility Functions**

Function	Return Value
`ArrayAppend(array, value)`	Places *`value`* in the (array length plus 1) element of the specified `array`. Returns `true` if successful.
`ArrayClear(array)`	Deletes all values from the specified *`array`*. Returns `true` if successful.
`ArrayDeleteAt(array, index)`	Deletes the value at *`index`* in the specified *`array`*. All elements with an index greater than *`index`* will be offset by `-1`. Returns `true` if successful. Throws `InvalidArrayIndexException` if *`index`* is not a valid index for *`array`*.
`ArrayInsertAt (array, index, value)`	Inserts *`value`* at *`index`* of *`array`*. The index for elements with an index greater than *`index`* will be offset by 1. Returns `true` if successful. Throws the `InvalidArrayIndexException` exception if *`index`* is not a valid index for *`array`*.
`ArrayNew(dimension)`	Creates an array of the specified *`dimension`*, which must be `1`, `2`, or `3`.
`ArrayPrepend(array, value)`	Places *`value`* in first element of specified the *`array`*. All indexes are offset by `1`. Returns `true` if successful.

Table 3.2 **Continued**

`ArrayResize(array, size)`	Resets *`array`* length to the specified *`size`*. Returns `true` if successful.
`ArraySet(array, startIndex, endIndex, value)`	Sets elements from index *`startIndex`* to *`endIndex`* to specified *`value`* in *`array`*. Returns `true` if successful.
`ArraySort(array, sortType [,order])`	Sorts all elements in *`array`* by *`sortType`* (`numeric`, `text`, or `textnocase`) in specified *`order`* (`asc` or `desc`). If *`array`* contains anything other than simple values, `ArraySort` throws `ArraySortSimpleValueException` exception. If array contains non-numeric values and *`sortType`* is numeric, `ArraySort` throws *`ValueNotNumeric`* exception. Returns `true` if successful.
`ArraySwap(array, index1, index2)`	Swaps the values at *`index1`* and *`index2`* of specified *`array`*. Throws `ArraySwapRangeException` exception if either *`index1`* or *`index2`* is an invalid index for *`array`*. Returns `true` if successful.
`ArrayToList(array [, delimiter])`	Returns a list delimited by *`delimiter`* (a comma by default) containing the values in *`array`*. Throws `ArrayNotOneDimensionException` exception if array contains more than one dimension.
`ListToArray(list [, delimiter])`	Returns an array populated with the contents of *`list`* using *`delimiter`* to specify indexes. Empty values in *`list`* are ignored.

3.11. Array Information Functions

You want to retrieve information about an array object.

Technique

Use ColdFusion array information functions to retrieve information about an array object.

```
<cfset aNewArray = ArrayNew(3)>
```

Comments

Array Information functions are useful for gathering information about array objects. There are three ColdFusion array information functions, described in detail in Table 3.3.

Table 3.3 **ColdFusion Array Utility Functions**

Function	Return Value
`ArrayIsEmpty(`*`array`*`)`	Determines whether the ColdFusion array object array is empty of elements. Returns true if empty.
ArrayLen(array)	Returns the number of elements found in *`array`*.
`IsArray(`*`array [, dimension`*`])`	Determines whether *`array`* is a ColdFusion array object. If *`dimension`* is defined, determines whether the element contained in *`dimension`* is a ColdFusion array object. Returns `true` if the item is an array.