F# - ARRAYS

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements that are all of the same type.

Creating Arrays

You can create arrays using various syntaxes and ways or by using the functions from the Array module. In this section, we will discuss creating arrays without using the module functions.

There are three syntactical ways of creating arrays without functions −

- By listing consecutive values between [| and |] and separated by semicolons.
- By putting each element on a separate line, in which case the semicolon separator is optional.
- By using sequence expressions.

You can access array elements by using a dot operator . and brackets [*and*].

The following example demonstrates creating arrays −

```
//using semicolon separator
let array1 = [| 1; 2; 3; 4; 5; 6 |]
for i in 0 .. array1.Length - 1 do
   printf "%d " array1.[i]
printfn" "

// without semicolon separator
let array2 =
   [|
      1
      2
      3
      4
      5
   |]
for i in 0 .. array2.Length - 1 do
   printf "%d " array2.[i]
printfn" "

//using sequence
let array3 = [| for i in 1 .. 10 -> i * i |]
for i in 0 .. array3.Length - 1 do
   printf "%d " array3.[i]
printfn" "
```

When you compile and execute the program, it yields the following output −

```
1 2 3 4 5 6
1 2 3 4 5
1 4 9 16 25 36 49 64 81 100
```

Basic Operations on Arrays

The library module Microsoft.FSharp.Collections.Array supports operations on one-dimensional arrays.

The following table shows the basic operations on Arrays −

Value	Description

append : 'T [] → 'T [] → 'T []	Creates an array that contains the elements of one array followed by the elements of another array.
average : ^T [] → ^T	Returns the average of the elements in an array.
averageBy : $'T \rightarrow^{U}$ → 'T [] → ^U	Returns the average of the elements generated by applying a function to each element of an array.
blit : 'T [] → int → 'T [] → int → int → unit	Reads a range of elements from one array and writes them into another.
choose : $'T \rightarrow Uoption$ → 'T [] → 'U []	Applies a supplied function to each element of an array. Returns an array that contains the results x for each element for which the function returns Some$x$.
collect : $'T \rightarrow 'U[]$ → T [] → 'U []	Applies the supplied function to each element of an array, concatenates the results, and returns the combined array.
concat : seq<'T []> → 'T []	Creates an array that contains the elements of each of the supplied sequence of arrays.
copy : 'T → 'T []	Creates an array that contains the elements of the supplied array.
create : int → 'T → 'T []	Creates an array whose elements are all initially the supplied value.
empty : 'T []	Returns an empty array of the given type.
exists : $'T \rightarrow bool$ → 'T [] → bool	Tests whether any element of an array satisfies the supplied predicate.
exists2 : $'T1 \rightarrow 'T2 \rightarrow bool$ → 'T1 [] → 'T2 [] → bool	Tests whether any pair of corresponding elements of two arrays satisfy the supplied condition.
fill : 'T [] → int → int → 'T → unit	Fills a range of elements of an array with the supplied value.
filter : $'T \rightarrow bool$ → 'T [] → 'T []	Returns a collection that contains only the elements of the supplied array for which the supplied condition returns **true**.
find : $'T \rightarrow bool$ → 'T [] → 'T	Returns the first element for which the supplied function returns **true**. Raises KeyNotFoundException if no such element exists.
findIndex : $'T \rightarrow bool$ → 'T [] → int	Returns the index of the first element in an array that satisfies the supplied condition. Raises KeyNotFoundException if none of the elements satisfy the condition.
fold : $'State \rightarrow 'T \rightarrow 'State$ → 'State → 'T [] → 'State	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f . . . (fsi0...) iN.
fold2 : $'State \rightarrow 'T1 \rightarrow 'T2 \rightarrow 'State$ → 'State → 'T1 [] → 'T2 [] → 'State	Applies a function to pairs of elements from two supplied arrays, left-to-right, threading an accumulator argument through the computation. The two input arrays must have the same lengths; otherwise, ArgumentException is

	raised.
foldBack : $'T \rightarrow 'State \rightarrow 'State$ → 'T [] → 'State → 'State	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f i0 . . . ($fiNs$).
foldBack2 : $'T1 \rightarrow 'T2 \rightarrow 'State \rightarrow 'State$ → 'T1 [] → 'T2 [] → 'State → 'State	Applies a function to pairs of elements from two supplied arrays, right-to-left, threading an accumulator argument through the computation. The two input arrays must have the same lengths; otherwise, ArgumentException is raised.
forall : $'T \rightarrow bool$ → 'T [] → bool	Tests whether all elements of an array satisfy the supplied condition.
forall2 : $'T1 \rightarrow 'T2 \rightarrow bool$ → 'T1 [] → 'T2 [] → bool	Tests whether all corresponding elements of two supplied arrays satisfy a supplied condition.
get : 'T [] → int → 'T	Gets an element from an array.
init : int → $int \rightarrow 'T$ → 'T []	Uses a supplied function to create an array of the supplied dimension.
isEmpty : 'T [] → bool	Tests whether an array has any elements.
iter : $'T \rightarrow unit$ → 'T [] → unit	Applies the supplied function to each element of an array.
iter2 : $'T1 \rightarrow 'T2 \rightarrow unit$ → 'T1 [] → 'T2 [] → unit)	Applies the supplied function to a pair of elements from matching indexes in two arrays. The two arrays must have the same lengths; otherwise, ArgumentException is raised.
iteri : $int \rightarrow 'T \rightarrow unit$ → 'T [] → unit	Applies the supplied function to each element of an array. The integer passed to the function indicates the index of the element.
iteri2 : $int \rightarrow 'T1 \rightarrow 'T2 \rightarrow unit$ → 'T1 [] → 'T2 [] → unit	Applies the supplied function to a pair of elements from matching indexes in two arrays, also passing the index of the elements. The two arrays must have the same lengths; otherwise, an ArgumentException is raised.
length : 'T [] → int	Returns the length of an array. The Length property does the same thing.
map : $'T \rightarrow 'U$ → 'T [] → 'U []	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.
map2 : $'T1 \rightarrow 'T2 \rightarrow 'U$ → 'T1 [] → 'T2 [] → 'U []	Creates an array whose elements are the results of applying the supplied function to the corresponding elements of two supplied arrays. The two input arrays must have the same lengths; otherwise, ArgumentException is raised.
mapi : $int \rightarrow 'T \rightarrow 'U$ → 'T [] → 'U []	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array. An integer index passed to the function indicates the index of the element being transformed.
mapi2 : $int \rightarrow 'T1 \rightarrow 'T2 \rightarrow 'U$ → 'T1 [] → 'T2 [] → 'U []	Creates an array whose elements are the results of applying the supplied function to the

	corresponding elements of the two collections pairwise, also passing the index of the elements. The two input arrays must have the same lengths; otherwise, ArgumentException is raised.
max : 'T [] → 'T	Returns the largest of all elements of an array. Operators.max is used to compare the elements.
maxBy : $'T \rightarrow 'U$ → 'T [] → 'T	Returns the largest of all elements of an array, compared via Operators.max on the function result.
min : ('T [] → 'T	Returns the smallest of all elements of an array. Operators.min is used to compare the elements.
minBy : $'T \rightarrow 'U$ → 'T [] → 'T	Returns the smallest of all elements of an array. Operators.min is used to compare the elements.
ofList : 'T list → 'T []	Creates an array from the supplied list.
ofSeq : seq<'T> → 'T []	Creates an array from the supplied enumerable object.
partition : $'T \rightarrow bool$ → 'T [] → 'T [] * 'T []	Splits an array into two arrays, one containing the elements for which the supplied condition returns **true,** and the other containing those for which it returns **false**.
permute : $int \rightarrow int$ → 'T [] → 'T []	Permutes the elements of an array according to the specified permutation.
pick : $'T \rightarrow 'Uoption$ → 'T [] → 'U	Applies the supplied function to successive elements of a supplied array, returning the first result where the function returns Some$x$ for some x. If the function never returns Some$x$, KeyNotFoundException is raised.
reduce : $'T \rightarrow 'T \rightarrow 'T$ → 'T [] → 'T	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f . . . ($fi0i1$...) iN. If the array has size zero, ArgumentException is raised.
reduceBack : $'T \rightarrow 'T \rightarrow 'T$ → 'T [] → 'T	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, this function computes f i0 . . . ($fiN-1iN$). If the array has size zero, ArgumentException is raised.
rev : 'T [] → 'T []	Reverses the order of the elements in a supplied array.
scan : $'State \rightarrow 'T \rightarrow 'State$ → 'State → 'T [] → 'State [])	Behaves like fold, but returns the intermediate results together with the final results.
scanBack : $'T \rightarrow 'State \rightarrow 'State$ → 'T [] → 'State → 'State []	Behaves like foldBack, but returns the intermediary results together with the final results.
set : 'T [] → int → 'T → unit	Sets an element of an array.
sort : 'T[] → 'T []	Sorts the elements of an array and returns a new array. Operators.compare is used to compare the elements.

sortBy : $'T \rightarrow 'Key$ → 'T [] → 'T []	Sorts the elements of an array by using the supplied function to transform the elements to the type on which the sort operation is based, and returns a new array. Operators.compare is used to compare the elements.
sortInPlace : 'T [] → unit	Sorts the elements of an array by changing the array in place, using the supplied comparison function. Operators.compare is used to compare the elements.
sortInPlaceBy : $'T \rightarrow 'Key$ → 'T [] → unit	Sorts the elements of an array by changing the array in place, using the supplied projection for the keys. Operators.compare is used to compare the elements.
sortInPlaceWith : $'T \rightarrow 'T \rightarrow int$ → 'T [] → unit	Sorts the elements of an array by using the supplied comparison function to change the array in place.
sortWith : $'T \rightarrow 'T \rightarrow int$ → 'T [] → 'T []	Sorts the elements of an array by using the supplied comparison function, and returns a new array.
sub : 'T [] → int → int → 'T []	Creates an array that contains the supplied subrange, which is specified by starting index and length.
sum : 'T [] → ^T	Returns the sum of the elements in the array.
sumBy : $'T \rightarrow U$ → 'T [] → ^U	Returns the sum of the results generated by applying a function to each element of an array.
toList : 'T [] → 'T list	Converts the supplied array to a list.
toSeq : 'T [] → seq<'T>	Views the supplied array as a sequence.
tryFind : $'T \rightarrow bool$ → 'T [] → 'T option	Returns the first element in the supplied array for which the supplied function returns **true**. Returns **None** if no such element exists.
tryFindIndex : $'T \rightarrow bool$ → 'T [] → int option	Returns the index of the first element in an array that satisfies the supplied condition.
tryPick : $'T \rightarrow 'Uoption$ → 'T [] → 'U option	Applies the supplied function to successive elements of the supplied array, and returns the first result where the function returns Some$x$ for some x. If the function never returns Some$x$, **None** is returned.
unzip : $'T1 * 'T2$ [] → 'T1 [] * 'T2 []	Splits an array of tuple pairs into a tuple of two arrays.
unzip3 : $'T1 * 'T2 * 'T3$ [] → 'T1 [] * 'T2 [] * 'T3 []	Splits an array of tuples of three elements into a tuple of three arrays.
zeroCreate : int → 'T []	Creates an array whose elements are initially set to the default value Unchecked.defaultof<'T>.
zip : 'T1 [] → 'T2 [] → $'T1 * 'T2$ []	Combines two arrays into an array of tuples that have two elements. The two arrays must have equal lengths; otherwise, ArgumentException is raised.
zip3 : 'T1 [] → 'T2 [] → 'T3 [] → $'T1 * 'T2 * 113'T3$ []	Combines three arrays into an array of tuples that have three elements. The three arrays must have equal lengths; otherwise,

ArgumentException is raised.

In the following section, we will see the uses of some of these functionalities.

Creating Arrays Using Functions

The Array module provides several functions that create an array from scratch.

- The **Array.empty** function creates a new empty array.
- The **Array.create** function creates an array of a specified size and sets all the elements to given values.
- The **Array.init** function creates an array, given a dimension and a function to generate the elements.
- The **Array.zeroCreate** function creates an array in which all the elements are initialized to the zero value.
- The **Array.copy** function creates a new array that contains elements that are copied from an existing array.
- The **Array.sub** function generates a new array from a subrange of an array.
- The **Array.append** function creates a new array by combining two existing arrays.
- The **Array.choose** function selects elements of an array to include in a new array.
- The **Array.collect** function runs a specified function on each array element of an existing array and then collects the elements generated by the function and combines them into a new array.
- The **Array.concat** function takes a sequence of arrays and combines them into a single array.
- The **Array.filter** function takes a Boolean condition function and generates a new array that contains only those elements from the input array for which the condition is true.
- The **Array.rev** function generates a new array by reversing the order of an existing array.

The following examples demonstrate these functions −

Example 1

```
(* using create and set *)
let array1 = Array.create 10 ""
for i in 0 .. array1.Length - 1 do
   Array.set array1 i (i.ToString())
for i in 0 .. array1.Length - 1 do
   printf "%s " (Array.get array1 i)
printfn " "

(* empty array *)
let array2 = Array.empty
printfn "Length of empty array: %d" array2.Length

let array3 = Array.create 10 7.0
printfn "Float Array: %A" array3

(* using the init and zeroCreate *)
let array4 = Array.init 10 (fun index -> index * index)
printfn "Array of squares: %A" array4

let array5 : float array = Array.zeroCreate 10
let (myZeroArray : float array) = Array.zeroCreate 10
printfn "Float Array: %A" array5
```

When you compile and execute the program, it yields the following output −

```
0 1 2 3 4 5 6 7 8 9
Length of empty array: 0
Float Array: [|7.0; 7.0; 7.0; 7.0; 7.0; 7.0; 7.0; 7.0; 7.0; 7.0|]
Array of squares: [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
Float Array: [|0.0; 0.0; 0.0; 0.0; 0.0; 0.0; 0.0; 0.0; 0.0; 0.0|]
```

Example 2

```
(* creating subarray from element 5 *)
(* containing 15 elements thereon *)

let array1 = [| 0 .. 50 |]
let array2 = Array.sub array1 5 15
printfn "Sub Array:"
printfn "%A" array2

(* appending two arrays *)
let array3 = [| 1; 2; 3; 4|]
let array4 = [| 5 .. 9 |]
printfn "Appended Array:"
let array5 = Array.append array3 array4
printfn "%A" array5

(* using the Choose function *)
let array6 = [| 1 .. 20 |]
let array7 = Array.choose (fun elem -> if elem % 3 = 0 then
                                          Some(float (elem))
                                       else
                                          None) array6
printfn "Array with Chosen elements:"
printfn "%A" array7

(*using the Collect function *)
let array8 = [| 2 .. 5 |]
let array9 = Array.collect (fun elem -> [| 0 .. elem - 1 |]) array8
printfn "Array with collected elements:"
printfn "%A" array9
```

When you compile and execute the program, it yields the following output −

```
Sub Array:
[|5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19|]
Appended Array:
[|1; 2; 3; 4; 5; 6; 7; 8; 9|]
Array with Chosen elements:
[|3.0; 6.0; 9.0; 12.0; 15.0; 18.0|]
Array with collected elements:
[|0; 1; 0; 1; 2; 0; 1; 2; 3; 0; 1; 2; 3; 4|]
```

Searching Arrays

The **Array.find** function takes a Boolean function and returns the first element for which the function returns true, else raises a KeyNotFoundException.

The **Array.findIndex** function works similarly except that it returns the index of the element instead of the element itself.

The following example demonstrates this.

Microsoft provides this interesting program example, which finds the first element in the range of a given number that is both a perfect square as well as a perfect cube −

```
let array1 = [| 2 .. 100 |]
```

```
let delta = 1.0e-10
let isPerfectSquare (x:int) =
   let y = sqrt (float x)
   abs(y - round y) < delta

let isPerfectCube (x:int) =
   let y = System.Math.Pow(float x, 1.0/3.0)
   abs(y - round y) < delta

let element = Array.find (fun elem -> isPerfectSquare elem && isPerfectCube elem)
array1

let index = Array.findIndex (fun elem -> isPerfectSquare elem && isPerfectCube elem)
array1

printfn "The first element that is both a square and a cube is %d and its index is %d."
element index
```

When you compile and execute the program, it yields the following output –

```
The first element that is both a square and a cube is 64 and its index is 62.
```