F# - LISTS

In F#, a list is an ordered, immutable series of elements of the same type. It is to some extent equivalent to a linked list data structure.

The F# module, **Microsoft.FSharp.Collections.List,** has the common operations on lists. However F# imports this module automatically and makes it accessible to every F# application.

Creating and Initializing a List

Following are the various ways of creating lists −

- Using list **literals**.
- Using **cons** :: operator.
- Using the **List.init** method of List module.
- Using some **syntactic constructs** called **List Comprehensions**.

List Literals

In this method, you just specify a semicolon-delimited sequence of values in square brackets. For example −

```
let list1 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

The cons :: Operator

With this method, you can add some values by prepending or **cons-ing** it to an existing list using the :: operator. For example −

```
let list1 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

[] denotes an empty list.

List init Method

The List.init method of the List module is often used for creating lists. This method has the type −

```
val init : int -> (int -> 'T) -> 'T list
```

The first argument is the desired length of the new list, and the second argument is an initializer function, which generates items in the list.

For example,

```
let list5 = List.init 5 (fun index -> (index, index * index, index * index * index))
```

Here, the index function generates the list.

List Comprehensions

List comprehensions are special syntactic constructs used for generating lists.

F# list comprehension syntax comes in two forms − ranges and generators.

Ranges have the constructs − [start .. end] and [start .. step .. end]

For example,

```
let list3 = [1 .. 10]
```

Generators have the construct – [for x in collection do ... yield expr]

For example,

```
let list6 = [ for a in 1 .. 10 do yield (a * a) ]
```

As the **yield** keyword pushes a single value into a list, the keyword, **yield!,** pushes a collection of values into the list.

The following function demonstrates the above methods –

Example

```
(* using list literals *)
let list1 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
printfn "The list: %A" list1

(*using cons operator *)
let list2 = 1 :: 2 :: 3 :: []
printfn "The list: %A" list2

(* using range constructs*)
let list3 = [1 .. 10]
printfn "The list: %A" list3

(* using range constructs *)
let list4 = ['a' .. 'm']
printfn "The list: %A" list4

(* using init method *)
let list5 = List.init 5 (fun index -> (index, index * index, index * index * index))
printfn "The list: %A" list5

(* using yield operator *)
let list6 = [ for a in 1 .. 10 do yield (a * a) ]
printfn "The list: %A" list6

(* using yield operator *)
let list7 = [ for a in 1 .. 100 do if a % 3 = 0 && a % 5 = 0 then yield a]
printfn "The list: %A" list7

(* using yield! operator *)
let list8 = [for a in 1 .. 3 do yield! [ a .. a + 3 ] ]
printfn "The list: %A" list8
```

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

```
The list: [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
The list: [1; 2; 3]
The list: [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
The list: ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'; 'h'; 'i'; 'j'; 'k'; 'l'; 'm']
The list: [(0, 0, 0); (1, 1, 1); (2, 4, 8); (3, 9, 27); (4, 16, 64)]
The list: [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
The list: [15; 30; 45; 60; 75; 90]
The list: [1; 2; 3; 4; 2; 3; 4; 5; 3; 4; 5; 6]
```

Properties of List Data Type

The following table shows various properties of list data type –

Property	Type	Description

Head	'T	The first element.
Empty	'T list	A static property that returns an empty list of the appropriate type.
IsEmpty	bool	**true** if the list has no elements.
Item	'T	The element at the specified index $zero - based$.
Length	int	The number of elements.
Tail	'T list	The list without the first element.

The following example shows the use of these properties −

Example

```
let list1 = [ 2; 4; 6; 8; 10; 12; 14; 16 ]

// Use of Properties
printfn "list1.IsEmpty is %b" (list1.IsEmpty)
printfn "list1.Length is %d" (list1.Length)
printfn "list1.Head is %d" (list1.Head)
printfn "list1.Tail.Head is %d" (list1.Tail.Head)
printfn "list1.Tail.Tail.Head is %d" (list1.Tail.Tail.Head)
printfn "list1.Item(1) is %d" (list1.Item(1))
```

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

```
list1.IsEmpty is false
list1.Length is 8
list1.Head is 2
list1.Tail.Head is 4
list1.Tail.Tail.Head is 6
list1.Item(1) is 4
```

Basic Operators on List

The following table shows the basic operations on list data type −

Value	Description
append : 'T list → 'T list → 'T list	Returns a new list that contains the elements of the first list followed by elements of the second.
average : 'T list → ^T	Returns the average of the elements in the list.
averageBy : $'T \rightarrow^{U}$ → 'T list → ^U	Returns the average of the elements generated by applying the function to each element of the list.
choose : $'T \rightarrow' Uoption$ → 'T list → 'U list	Applies the given function to each element of the list. Returns the list comprised of the results for each element where the function returns **Some**.
collect : $'T \rightarrow' Ulist$ → 'T list → 'U list	For each element of the list, applies the given function. Concatenates all the results and return the combined list.
concat : seq<'T list> → 'T list	Returns a new list that contains the elements of each the lists in order.
empty : 'T list	Returns an empty list of the given type.

exists : $'T \rightarrow bool$ → 'T list → bool	Tests if any element of the list satisfies the given predicate.
exists2 : $'T1 \rightarrow 'T2 \rightarrow bool$ → 'T1 list → 'T2 list → bool	Tests if any pair of corresponding elements of the lists satisfies the given predicate.
filter : $'T \rightarrow bool$ → 'T list → 'T list	Returns a new collection containing only the elements of the collection for which the given predicate returns **true**.
find : $'T \rightarrow bool$ → 'T list → 'T	Returns the first element for which the given function returns **true**.
findIndex : $'T \rightarrow bool$ → 'T list → int	Returns the index of the first element in the list that satisfies the given predicate.
fold : $'State \rightarrow 'T \rightarrow 'State$ → 'State → 'T list → 'State	Applies a function to each element of the collection, threading an accumulator argument through the computation. This function takes the second argument, and applies the function to it and the first element of the list. Then, it passes this result into the function along with the second element, and so on. Finally, it returns the final result. If the input function is f and the elements are i0...iN, then this function computes f . . . ($fsi0$ i1 ...) iN.
fold2 : $'State \rightarrow 'T1 \rightarrow 'T2 \rightarrow 'State$ → 'State → 'T1 list → 'T2 list → 'State	Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation. The collections must have identical sizes. If the input function is f and the elements are i0...iN and j0...jN, then this function computes f . . . ($fsi0j0$...) iN jN.
foldBack : $'T \rightarrow 'State \rightarrow 'State$ → 'T list → 'State → 'State	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function isf and the elements are i0...iN then computes f i0 . . . ($fiNs$).
foldBack2 : $'T1 \rightarrow 'T2 \rightarrow 'State \rightarrow 'State$ → 'T1 list → 'T2 list → 'State → 'State	Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation. The collections must have identical sizes. If the input function is f and the elements are i0...iN and j0...jN, then this function computes f i0 j0 . . . ($fiNjNs$).
forall : $'T \rightarrow bool$ → 'T list → bool	Tests if all elements of the collection satisfy the given predicate.
forall2 : $'T1 \rightarrow 'T2 \rightarrow bool$ → 'T1 list → 'T2 list → bool	Tests if all corresponding elements of the collection satisfy the given predicate pairwise.
head : 'T list → 'T	Returns the first element of the list.
init : int → $int \rightarrow 'T$ → 'T list	Creates a list by calling the given generator on each index.
isEmpty : 'T list → bool	Returns **true** if the list contains no elements, **false** otherwise.
iter : $'T \rightarrow unit$ → 'T list → unit	Applies the given function to each element of the collection.
iter2 : $'T1 \rightarrow 'T2 \rightarrow unit$ → 'T1 list → 'T2 list →	Applies the given function to two collections

unit	simultaneously. The collections must have identical size.
iteri : $int \rightarrow 'T \rightarrow unit$ → 'T list → unit	Applies the given function to each element of the collection. The integer passed to the function indicates the index of element.
iteri2 : $int \rightarrow 'T1 \rightarrow 'T2 \rightarrow unit$ → 'T1 list → 'T2 list → unit	Applies the given function to two collections simultaneously. The collections must have identical size. The integer passed to the function indicates the index of element.
length : 'T list → int	Returns the length of the list.
map : $'T \rightarrow 'U$ → 'T list → 'U list	Creates a new collection whose elements are the results of applying the given function to each of the elements of the collection.
map2 : $'T1 \rightarrow 'T2 \rightarrow 'U$ → 'T1 list → 'T2 list → 'U list	Creates a new collection whose elements are the results of applying the given function to the corresponding elements of the two collections pairwise.
map3 : $'T1 \rightarrow 'T2 \rightarrow 'T3 \rightarrow 'U$ → 'T1 list → 'T2 list → 'T3 list → 'U list	Creates a new collection whose elements are the results of applying the given function to the corresponding elements of the three collections simultaneously.
mapi : $int \rightarrow 'T \rightarrow 'U$ → 'T list → 'U list	Creates a new collection whose elements are the results of applying the given function to each of the elements of the collection. The integer index passed to the function indicates the index $from 0$ of element being transformed.
mapi2 : $int \rightarrow 'T1 \rightarrow 'T2 \rightarrow 'U$ → 'T1 list → 'T2 list → 'U list	Like List.mapi, but mapping corresponding elements from two lists of equal length.
max : 'T list → 'T	Returns the greatest of all elements of the list, compared by using Operators.max.
maxBy : $'T \rightarrow 'U$ → 'T list → 'T	Returns the greatest of all elements of the list, compared by using Operators.max on the function result.
min : 'T list → 'T	Returns the lowest of all elements of the list, compared by using Operators.min.
minBy : $'T \rightarrow 'U$ → 'T list → 'T	Returns the lowest of all elements of the list, compared by using Operators.min on the function result
nth : 'T list → int → 'T	Indexes into the list. The first element has index 0.
ofArray : 'T [] → 'T list	Creates a list from the given array.
ofSeq : seq<'T> → 'T list	Creates a new list from the given enumerable object.
partition : $'T \rightarrow bool$ → 'T list * 'T list	Splits the collection into two collections, containing the elements for which the given predicate returns **true** and **false** respectively.
permute : $int \rightarrow int$ → 'T list → 'T list	Returns a list with all elements permuted according to the specified permutation.
pick : $'T \rightarrow 'Uoption$ → 'T list → 'U	Applies the given function to successive

	elements, returning the first result where function returns **Some** for some value.
reduce : $'T \rightarrow 'T \rightarrow 'T$ → 'T list → 'T	Applies a function to each element of the collection, threading an accumulator argument through the computation. This function applies the specified function to the first two elements of the list. It then passes this result into the function along with the third element, and so on. Finally, it returns the final result. If the input function is f and the elements are i0...iN, then this function computes f . . . ($fi0i1$ i2 ...) iN.
reduceBack : $'T \rightarrow 'T \rightarrow 'T$ → 'T list → 'T	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function isf and the elements are i0...iN, then this function computes f i0 . . . ($fiN-1iN$).
replicate : $int \rightarrow 'T \rightarrow 'Tlist$	Creates a list by calling the given generator on each index.
rev : 'T list → 'T list	Returns a new list with the elements in reverse order.
scan : $'State \rightarrow 'T \rightarrow 'State$ → 'State → 'T list → 'State list	Applies a function to each element of the collection, threading an accumulator argument through the computation. This function takes the second argument, and applies the specified function to it and the first element of the list. Then, it passes this result into the function along with the second element and so on. Finally, it returns the list of intermediate results and the final result.
scanBack : $'T \rightarrow 'State \rightarrow 'State$ → 'T list → 'State → 'State list	Like foldBack, but returns both the intermediate and final results
sort : 'T list → 'T list	Sorts the given list using Operators.compare.
sortBy : $'T \rightarrow 'Key$ → 'T list → 'T list	Sorts the given list using keys given by the given projection. Keys are compared using Operators.compare.
sortWith : $'T \rightarrow 'T \rightarrow int$ → 'T list → 'T list	Sorts the given list using the given comparison function.
sum : ^T list → ^T	Returns the sum of the elements in the list.
sumBy : $'T \rightarrow ^U$ → 'T list → ^U	Returns the sum of the results generated by applying the function to each element of the list.
tail : 'T list → 'T list	Returns the input list without the first element.
toArray : 'T list → 'T []	Creates an array from the given list.
toSeq : 'T list → seq<'T>	Views the given list as a sequence.
tryFind : $'T \rightarrow bool$ → 'T list → 'T option	Returns the first element for which the given function returns **true**. Return **None** if no such element exists.
tryFindIndex : $'T \rightarrow bool$ → 'T list → int option	Returns the index of the first element in the list that satisfies the given predicate. Return **None** if no such element exists.
tryPick : $'T \rightarrow 'Uoption$ → 'T list → 'U option	Applies the given function to successive

	elements, returning the first result where function returns **Some** for some value. If no such element exists then return **None**.
unzip : $'T1 * 'T2$ list → 'T1 list * 'T2 list	Splits a list of pairs into two lists.
unzip3 : $'T1 * 'T2 * 'T3$ list → 'T1 list * 'T2 list * 'T3 list	Splits a list of triples into three lists.
zip : 'T1 list → 'T2 list → $'T1 * 'T2$ list	Combines the two lists into a list of pairs. The two lists must have equal lengths.
zip3 : 'T1 list → 'T2 list → 'T3 list → $'T1 * 'T2 * 'T3$ list	Combines the three lists into a list of triples. The lists must have equal lengths.

The following examples demonstrate the uses of the above functionalities −

Example 1

This program shows reversing a list recursively −

```
let list1 = [ 2; 4; 6; 8; 10; 12; 14; 16 ]
printfn "The original list: %A" list1

let reverse lt =
   let rec loop acc = function
      | [] -> acc
      | hd :: tl -> loop (hd :: acc) tl
   loop [] lt

printfn "The reversed list: %A" (reverse list1)
```

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

```
The original list: [2; 4; 6; 8; 10; 12; 14; 16]
The reversed list: [16; 14; 12; 10; 8; 6; 4; 2]
```

However, you can use the **rev** function of the module for the same purpose −

```
let list1 = [ 2; 4; 6; 8; 10; 12; 14; 16 ]
printfn "The original list: %A" list1
printfn "The reversed list: %A" (List.rev list1)
```

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

```
The original list: [2; 4; 6; 8; 10; 12; 14; 16]
The reversed list: [16; 14; 12; 10; 8; 6; 4; 2]
```

Example 2

This program shows filtering a list using the **List.filter** method −

```
let list1 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
printfn "The list: %A" list1
let list2 = list1 |> List.filter (fun x -> x % 2 = 0);;
printfn "The Filtered list: %A" list2
```

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

```
The list: [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
The Filtered list: [2; 4; 6; 8; 10]
```

Example 3

The **List.map** method maps a list from one type to another −

```
let list1 = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
printfn "The list: %A" list1
let list2 = list1 |> List.map (fun x -> (x * x).ToString());;
printfn "The Mapped list: %A" list2
```

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

```
The list: [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
The Mapped list: ["1"; "4"; "9"; "16"; "25"; "36"; "49"; "64"; "81"; "100"]
```

Example 4

The **List.append** method and the @ operator appends one list to another −

```
let list1 = [1; 2; 3; 4; 5 ]
let list2 = [6; 7; 8; 9; 10]
let list3 = List.append list1 list2

printfn "The first list: %A" list1
printfn "The second list: %A" list2
printfn "The appened list: %A" list3

let lt1 = ['a'; 'b';'c' ]
let lt2 = ['e'; 'f';'g' ]
let lt3 = lt1 @ lt2

printfn "The first list: %A" lt1
printfn "The second list: %A" lt2
printfn "The appened list: %A" lt3
```

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

```
The first list: [1; 2; 3; 4; 5]
The second list: [6; 7; 8; 9; 10]
The appened list: [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
The first list: ['a'; 'b'; 'c']
The second list: ['e'; 'f'; 'g']
The appened list: ['a'; 'b'; 'c'; 'e'; 'f'; 'g']
```

Example 5

The **List.sort** method sorts a list. The **List.sum** method gives the sum of elements in the list and the **List.average** method gives the average of elements in the list −

```
let list1 = [9.0; 0.0; 2.0; -4.5; 11.2; 8.0; -10.0]
printfn "The list: %A" list1

let list2 = List.sort list1
printfn "The sorted list: %A" list2

let s = List.sum list1
let avg = List.average list1
printfn "The sum: %f" s
printfn "The average: %f" avg
```

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

```
The list: [9.0; 0.0; 2.0; -4.5; 11.2; 8.0; -10.0]
The sorted list: [-10.0; -4.5; 0.0; 2.0; 8.0; 9.0; 11.2]
The sum: 15.700000
```

```
The average: 2.242857
```

A "fold" operation applies a function to each element in a list, aggregates the result of the function in an accumulator variable, and returns the accumulator as the result of the fold operation.

Example 6

The **List.fold** method applies a function to each element from left to right, while **List.foldBack** applies a function to each element from right to left.

```
let sumList list = List.fold (fun acc elem -> acc + elem) 0 list
printfn "Sum of the elements of list %A is %d." [ 1 .. 10 ] (sumList [ 1 .. 10 ])
```

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

```
Sum of the elements of list [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] is 55.
```