F# - SETS

A set in F# is a data structure that acts as a collection of items without preserving the order in which items are inserted. Sets do not allow duplicate entries to be inserted into the collection.

Creating Sets

Sets can be created in the following ways −

- By creating an empty set using Set.empty and adding items using the add function.
- Converting sequences and lists to sets.

The following program demonstrates the techniques −

```
(* creating sets *)
let set1 = Set.empty.Add(3).Add(5).Add(7). Add(9)
printfn"The new set: %A" set1

let weekdays = Set.ofList ["mon"; "tues"; "wed"; "thurs"; "fri"]
printfn "The list set: %A" weekdays

let set2 = Set.ofSeq [ 1 .. 2.. 10 ]
printfn "The sequence set: %A" set2
```

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

```
The new set: set [3; 5; 7; 9]
The list set: set ["fri"; "mon"; "thurs"; "tues"; "wed"]
The sequence set: set [1; 3; 5; 7; 9]
```

Basic Operations on Sets

The following table shows the basic operations on sets −

Value	Description
add : 'T → Set<'T> → Set<'T>	Returns a new set with an element added to the set. No exception is raised if the set already contains the given element.
contains : 'T → Set<'T> → bool	Evaluates to **true** if the given element is in the given set.
count : Set<'T> → int	Returns the number of elements in the set.
difference : Set<'T> → Set<'T> → Set<'T>	Returns a new set with the elements of the second set removed from the first.
empty : Set<'T>	The empty set for the specified type.
exists : $'T \rightarrow bool$ → Set<'T> → bool	Tests if any element of the collection satisfies the given predicate. If the input function is predicate and the elements are i0...iN, then this function computes predicate i0 or ... or predicate iN.
filter : $'T \rightarrow bool$ → Set<'T> → Set<'T>	Returns a new collection containing only the elements of the collection for which the given predicate returns **true**.
fold : $'State \rightarrow 'T \rightarrow 'State$ → 'State → Set<'T> →	Applies the given accumulating function to all

'State	the elements of the set.
foldBack : $'T \rightarrow 'State \rightarrow 'State$ → Set<'T> → 'State → 'State	Applies the given accumulating function to all the elements of the set.
forall : $'T \rightarrow bool$ → Set<'T> → bool	Tests if all elements of the collection satisfy the given predicate. If the input function is p and the elements are i0...iN, then this function computes p i0 && ... && p iN.
intersect : Set<'T> → Set<'T> → Set<'T>	Computes the intersection of the two sets.
intersectMany : seq<Set<'T>> → Set<'T>	Computes the intersection of a sequence of sets. The sequence must be non-empty.
isEmpty : Set<'T> → bool	Returns **true** if the set is empty.
isProperSubset : Set<'T> → Set<'T> → bool	Evaluates to **true** if all elements of the first set are in the second, and at least one element of the second is not in the first.
isProperSuperset : Set<'T> → Set<'T> → bool	Evaluates to **true** if all elements of the second set are in the first, and at least one element of the first is not in the second.
isSubset : Set<'T> → Set<'T> → bool	Evaluates to **true** if all elements of the first set are in the second.
isSuperset : Set<'T> → Set<'T> → bool	Evaluates to **true** if all elements of the second set are in the first.
iter : $'T \rightarrow unit$ → Set<'T> → unit	Applies the given function to each element of the set, in order according to the comparison function.
map : $'T \rightarrow 'U$ → Set<'T> → Set<'U>	Returns a new collection containing the results of applying the given function to each element of the input set.
maxElement : Set<'T> → 'T	Returns the highest element in the set according to the ordering being used for the set.
minElement : Set<'T> → 'T	Returns the lowest element in the set according to the ordering being used for the set.
ofArray : 'T array → Set<'T>	Creates a set that contains the same elements as the given array.
ofList : 'T list → Set<'T>	Creates a set that contains the same elements as the given list.
ofSeq : seq<'T> → Set<'T>	Creates a new collection from the given enumerable object.
partition : $'T \rightarrow bool$ → Set<'T> → Set<'T> * Set<'T>	Splits the set into two sets containing the elements for which the given predicate returns true and false respectively.
remove : 'T → Set<'T> → Set<'T>	Returns a new set with the given element removed. No exception is raised if the set doesn't contain the given element.
singleton : 'T → Set<'T>	The set containing the given element.
toArray : Set<'T> → 'T array	Creates an array that contains the elements of the set in order.

toList : Set<'T> → 'T list	Creates a list that contains the elements of the set in order.
toSeq : Set<'T> → seq<'T>	Returns an ordered view of the collection as an enumerable object.
union : Set<'T> → Set<'T> → Set<'T>	Computes the union of the two sets.
unionMany : seq<Set<'T>> → Set<'T>	Computes the union of a sequence of sets.

The following example demonstrates the uses of some of the above functionalities −

Example

```
let a = Set.ofSeq [ 1 ..2.. 20 ]
let b = Set.ofSeq [ 1 ..3 .. 20 ]
let c = Set.intersect a b
let d = Set.union a b
let e = Set.difference a b

printfn "Set a: "
Set.iter (fun x -> printf "%O " x) a
printfn""

printfn "Set b: "
Set.iter (fun x -> printf "%O " x) b
printfn""

printfn "Set c = set intersect of a and b : "
Set.iter (fun x -> printf "%O " x) c
printfn""

printfn "Set d = set union of a and b : "
Set.iter (fun x -> printf "%O " x) d
printfn""

printfn "Set e = set difference of a and b : "
Set.iter (fun x -> printf "%O " x) e
printfn""
```

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

```
Set a:
1 3 5 7 9 11 13 15 17 19
Set b:
1 4 7 10 13 16 19
Set c = set intersect of a and b :
1 7 13 19
Set d = set union of a and b :
1 3 4 5 7 9 10 11 13 15 16 17 19
Set e = set difference of a and b :
3 5 9 11 15 17
```