F# - INTERFACES

Interfaces provide an abstract way of writing up the implementation details of a class. It is a template that declares the methods the class must implement and expose publicly.

Syntax

An interface specifies the sets of related members that other classes implement. It has the following syntax −

```
// Interface declaration:
[ attributes ]
type interface-name =
   [ interface ]
      [ inherit base-interface-name ...]
      abstract member1 : [ argument-types1 -> ] return-type1
      abstract member2 : [ argument-types2 -> ] return-type2
      ...
   [ end ]

// Implementing, inside a class type definition:
interface interface-name with
   member self-identifier.member1 argument-list = method-body1
   member self-identifier.member2 argument-list = method-body2
// Implementing, by using an object expression:
[ attributes ]
let class-name (argument-list) =
   { new interface-name with
      member self-identifier.member1 argument-list = method-body1
      member self-identifier.member2 argument-list = method-body2
      [ base-interface-definitions ]
   }
member-list
```

Please note −

- In an interface declaration the members are not implemented.
- The members are abstract, declared by the **abstract** keyword. However you may provide a default implementation using the **default** keyword.
- You can implement interfaces either by using object expressions or by using class types.
- In class or object implementation, you need to provide method bodies for abstract methods of the interface.
- The keywords **interface** and **end,** which mark the start and end of the definition, are optional.

For example,

```
type IPerson =
   abstract Name : string
   abstract Enter : unit -> unit
   abstract Leave : unit -> unit
```

Calling Interface Methods

Interface methods are called through the interface, not through the instance of the class or type implementing interface. To call an interface method, you up cast to the interface type by using the **:>** operator *upcastoperator*.

For example,

```
(s :> IPerson).Enter()
(s :> IPerson).Leave()
```

The following example illustrates the concept −

Example

```
type IPerson =
   abstract Name : string
   abstract Enter : unit -> unit
   abstract Leave : unit -> unit

type Student(name : string, id : int) =
   member this.ID = id
   interface IPerson with
      member this.Name = name
      member this.Enter() = printfn "Student entering premises!"
      member this.Leave() = printfn "Student leaving premises!"

type StuffMember(name : string, id : int, salary : float) =
   let mutable _salary = salary

   member this.Salary
      with get() = _salary
      and set(value) = _salary <- value

   interface IPerson with
      member this.Name = name
      member this.Enter() = printfn "Stuff member entering premises!"
      member this.Leave() = printfn "Stuff member leaving premises!"

let s = new Student("Zara", 1234)
let st = new StuffMember("Rohit", 34, 50000.0)

(s :> IPerson).Enter()
(s :> IPerson).Leave()
(st :> IPerson).Enter()
(st :> IPerson).Leave()
```

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

```
Student entering premises!
Student leaving premises!
Stuff member entering premises!
Stuff member leaving premises!
```

Interface Inheritance

Interfaces can inherit from one or more base interfaces.

The following example shows the concept −

```
type Interface1 =
   abstract member doubleIt: int -> int

type Interface2 =
   abstract member tripleIt: int -> int

type Interface3 =
   inherit Interface1
   inherit Interface2
   abstract member printIt: int -> string

type multiplierClass() =
   interface Interface3 with
      member this.doubleIt(a) = 2 * a
```

```
      member this.tripleIt(a) = 3 * a
      member this.printIt(a) = a.ToString()

let ml = multiplierClass()
printfn "%d" ((ml:>Interface3).doubleIt(5))
printfn "%d" ((ml:>Interface3).tripleIt(5))
printfn "%s" ((ml:>Interface3).printIt(5))
```

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

```
10
15
5
```