F# - INHERITANCE

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the IS-A relationship. For example, mammal IS A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base Class and Sub Class

A subclass is derived from a base class, which is already defined. A subclass inherits the members of the base class, as well as has its own members.

A subclass is defined using the **inherit** keyword as shown below −

```
type MyDerived(...) =
   inherit MyBase(...)
```

In F#, a class can have at most one direct base class. If you do not specify a base class by using the **inherit** keyword, the class implicitly inherits from Object.

Please note −

- The methods and members of the base class are available to users of the derived class like the direct members of the derived class.
- Let bindings and constructor parameters are private to a class and, therefore, cannot be accessed from derived classes.
- The keyword **base** refers to the base class instance. It is used like the self-identifier.

Example

```
type Person(name) =
   member x.Name = name
   member x.Greet() = printfn "Hi, I'm %s" x.Name

type Student(name, studentID : int) =
   inherit Person(name)
   let mutable _GPA = 0.0
   member x.StudentID = studentID
   member x.GPA
      with get() = _GPA
      and set value = _GPA <- value

type Teacher(name, expertise : string) =
   inherit Person(name)

   let mutable _salary = 0.0
   member x.Salary
      with get() = _salary
      and set value = _salary <- value
   member x.Expertise = expertise

//using the subclasses
let p = new Person("Mohan")
let st = new Student("Zara", 1234)
```

```
let tr = new Teacher("Mariam", "Java")

p.Greet()
st.Greet()
tr.Greet()
```

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

```
Hi, I'm Mohan
Hi, I'm Zara
Hi, I'm Mariam
```

Overriding Methods

You can override a default behavior of a base class method and implement it differently in the subclass or the derived class.

Methods in F# are not overridable by default.

To override methods in a derived class, you have to declare your method as overridable using the **abstract** and **default** keywords as follows −

```
type Person(name) =
   member x.Name = name
   abstract Greet : unit -> unit
   default x.Greet() = printfn "Hi, I'm %s" x.Name
```

Now, the *Greet* method of the Person class can be overridden in derived classes. The following example demonstrates this −

Example

```
type Person(name) =
   member x.Name = name
   abstract Greet : unit -> unit
   default x.Greet() = printfn "Hi, I'm %s" x.Name

type Student(name, studentID : int) =
   inherit Person(name)

   let mutable _GPA = 0.0

   member x.StudentID = studentID
   member x.GPA
      with get() = _GPA
      and set value = _GPA <- value
   override x.Greet() = printfn "Student %s" x.Name

type Teacher(name, expertise : string) =
   inherit Person(name)
   let mutable _salary = 0.0
   member x.Salary
      with get() = _salary
      and set value = _salary <- value

   member x.Expertise = expertise
   override x.Greet() = printfn "Teacher %s." x.Name

//using the subclasses
let p = new Person("Mohan")
let st = new Student("Zara", 1234)
let tr = new Teacher("Mariam", "Java")

//default Greet
p.Greet()
```

```
//Overriden Greet
st.Greet()
tr.Greet()
```

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

```
Hi, I'm Mohan
Student Zara
Teacher Mariam.
```

Abstract Class

At times you need to provide an incomplete implementation of an object, which should not be implemented in reality. Later, some other programmer should create subclasses of the abstract class to a complete implementation.

For example, the Person class will not be needed in a School Management System. However, the Student or the Teacher class will be needed. In such cases, you can declare the Person class as an abstract class.

The **AbstractClass** attribute tells the compiler that the class has some abstract members.

You cannot create an instance of an abstract class because the class is not fully implemented.

The following example demonstrates this −

Example

```
[<AbstractClass>]
type Person(name) =
   member x.Name = name
   abstract Greet : unit -> unit

type Student(name, studentID : int) =
   inherit Person(name)
   let mutable _GPA = 0.0
   member x.StudentID = studentID
   member x.GPA
      with get() = _GPA
      and set value = _GPA <- value
   override x.Greet() = printfn "Student %s" x.Name

type Teacher(name, expertise : string) =
   inherit Person(name)
   let mutable _salary = 0.0
   member x.Salary
      with get() = _salary
      and set value = _salary <- value
   member x.Expertise = expertise
   override x.Greet() = printfn "Teacher %s." x.Name

let st = new Student("Zara", 1234)
let tr = new Teacher("Mariam", "Java")

//Overriden Greet
st.Greet()
tr.Greet()
```

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

```
Student Zara
Teacher Mariam.
```