F# - QUICK GUIDE

F# - OVERVIEW

F# is a functional programming language. To understand F# constructs, you need to read a couple of lines about the programming paradigm named **Functional Programming**.

Functional programming treats computer programs as mathematical functions. In functional programming, the focus would be on constants and functions, instead of variables and states. Because functions and constants are things that don't change.

In functional programming, you will write modular programs, i.e., the programs would consist of functions that will take other functions as input.

Programs written in functional programming language tend to be concise.

About F#

Following are the basic information about F# −

- It was developed in 2005 at Microsoft Research.
- It is a part of Microsoft's family of .Net language.
- It is a functional programming language.
- It is based on the functional programming language OCaml.

Features of F#

- It is .Net implementation of OCaml.
- It compiles .Net CLI (Common Language Interface) byte code or MSIL (Microsoft Intermediate Language) that runs on CLR (Common Language Runtime).
- It provides type inference.
- It provides rich pattern matching constructs.
- It has interactive scripting and debugging capabilities.
- It allows writing higher order functions.
- It provides well developed object model.

Use of F#

F# is normally used in the following areas −

- Making scientific model
- Mathematical problem solving
- Artificial intelligence research work
- Financial modelling
- Graphic design
- CPU design
- Compiler programming
- Telecommunications

It is also used in CRUD apps, web pages, GUI games and other general purpose programs.

F# - ENVIRONMENT SETUP

The tools required for F# programming are discussed in this chapter.

Integrated Development Environment(IDE) for F#

Microsoft provides Visual Studio 2013 for F# programming.

The free Visual Studio 2013 Community Edition is available from Microsoft's official website. Visual Studio 2013 Community and above comes with the Visual F# Tools. The Visual F# Tools include the command-line compiler (fsc.exe) and F# Interactive (fsi.exe).

Using these tools, you can write all kinds of F# programs from simple command-line applications to more complex applications. You can also write F# source code files using a basic text editor, like Notepad, and compile the code into assemblies using the command-line compiler.

You can download it from Microsoft Visual Studio. It gets automatically installed in your machine.

Writing F# Programs On Links

Please visit the F# official website for the latest instructions on getting the tools as a Debian package or compiling them directly from the source − http://fsharp.org/use/linux/.

> ### *Try it Option Online*
>
> *We have set up the F# Programming environment online. You can easily compile and execute all the available examples online along with doing your theory work. It gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.*
>
> *Try the following example using the Try it option or use the url − http://www.compileonline.com/.*
>
> ```
> (* This is a comment *)
> (* Sample Hello World program using F# *)
> printfn "Hello World!"
> ```
>
> *For most of the examples given in this tutorial, you will find a Try it option in our website code sections at the top right corner that will take you to the online compiler. So just make use of it and enjoy your learning.*

F# - PROGRAM STRUCTURE

F# is a Functional Programming language.

In F#, functions work like data types. You can declare and use a function in the same way like any other variable.

In general, an F# application does not have any specific entry point. The compiler executes all top-level statements in the file from top to bottom.

However, to follow procedural programming style, many applications keep a single top level statement that calls the main loop.

The following code shows a simple F# program −

```
open System
(* This is a multi-line comment *)
// This is a single-line comment

let sign num =
   if num > 0 then "positive"
```

```
   elif num < 0 then "negative"
   else "zero"

let main() =
   Console.WriteLine("sign 5: {0}", (sign 5))

main()
```

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

```
sign 5: positive
```

Please note that −

- An F# code file might begin with a number of **open** statements that is used to import namespaces.
- The body of the files includes other functions that implement the business logic of the application.
- The main loop contains the top executable statements.

F# - BASIC SYNTAX

You have seen the basic structure of an F# program, so it will be easy to understand other basic building blocks of the F# programming language.

Tokens in F#

An F# program consists of various tokens. A token could be a keyword, an identifier, a constant, a string literal, or a symbol. We can categorize F# tokens into two types −

- Keywords
- Symbol and Operators

F# Keywords

The following table shows the keywords and brief descriptions of the keywords. We will discuss the use of these keywords in subsequent chapters.

Keyword	Description
abstract	Indicates a method that either has no implementation in the type in which it is declared or that is virtual and has a default implementation.
and	Used in mutually recursive bindings, in property declarations, and with multiple constraints on generic parameters.
as	Used to give the current class object an object name. Also used to give a name to a whole pattern within a pattern match.
assert	Used to verify code during debugging.
base	Used as the name of the base class object.
begin	In verbose syntax, indicates the start of a code block.
class	In verbose syntax, indicates the start of a class definition.
default	Indicates an implementation of an abstract method; used together with an abstract method declaration to create a virtual method.
delegate	Used to declare a delegate.

do	Used in looping constructs or to execute imperative code.
done	In verbose syntax, indicates the end of a block of code in a looping expression.
downcast	Used to convert to a type that is lower in the inheritance chain.
downto	In a **for** expression, used when counting in reverse.
elif	Used in conditional branching. A short form of else if.
else	Used in conditional branching.
end	In type definitions and type extensions, indicates the end of a section of member definitions. In verbose syntax, used to specify the end of a code block that starts with the begin keyword.
exception	Used to declare an exception type.
extern	Indicates that a declared program element is defined in another binary or assembly.
false	Used as a Boolean literal.
finally	Used together with try to introduce a block of code that executes regardless of whether an exception occurs.
for	Used in looping constructs.
fun	Used in lambda expressions, also known as anonymous functions.
function	Used as a shorter alternative to the fun keyword and a match expression in a lambda expression that has pattern matching on a single argument.
global	Used to reference the top-level .NET namespace.
if	Used in conditional branching constructs.
in	Used for sequence expressions and, in verbose syntax, to separate expressions from bindings.
inherit	Used to specify a base class or base interface.
inline	Used to indicate a function that should be integrated directly into the caller's code.
interface	Used to declare and implement interfaces.
internal	Used to specify that a member is visible inside an assembly but not outside it.
lazy	Used to specify a computation that is to be performed only when a result is needed.
let	Used to associate, or bind, a name to a value or function.
let!	Used in asynchronous workflows to bind a name to the result of an asynchronous computation, or, in other computation expressions, used to bind a name to a result, which is of the computation type.
match	Used to branch by comparing a value to a pattern.
member	Used to declare a property or method in an object type.
module	Used to associate a name with a group of related types, values, and functions, to logically separate it from other code.

mutable	Used to declare a variable, that is, a value that can be changed.
namespace	Used to associate a name with a group of related types and modules, to logically separate it from other code.
new	Used to declare, define, or invoke a constructor that creates or that can create an object. Also used in generic parameter constraints to indicate that a type must have a certain constructor.
not	Not actually a keyword. However, not struct in combination is used as a generic parameter constraint.
null	Indicates the absence of an object. Also used in generic parameter constraints.
of	Used in discriminated unions to indicate the type of categories of values, and in delegate and exception declarations.
open	Used to make the contents of a namespace or module available without qualification.
or	Used with Boolean conditions as a Boolean or operator. Equivalent to ||. Also used in member constraints.
override	Used to implement a version of an abstract or virtual method that differs from the base version.
private	Restricts access to a member to code in the same type or module.
public	Allows access to a member from outside the type.
rec	Used to indicate that a function is recursive.
return	Used to indicate a value to provide as the result of a computation expression.
return!	Used to indicate a computation expression that, when evaluated, provides the result of the containing computation expression.
select	Used in query expressions to specify what fields or columns to extract. Note that this is a contextual keyword, which means that it is not actually a reserved word and it only acts like a keyword in appropriate context.
static	Used to indicate a method or property that can be called without an instance of a type, or a value member that is shared among all instances of a type.
struct	Used to declare a structure type. Also used in generic parameter constraints. Used for OCaml compatibility in module definitions.
then	Used in conditional expressions. Also used to perform side effects after object construction.

to	Used in for loops to indicate a range.
true	Used as a Boolean literal.
try	Used to introduce a block of code that might generate an exception. Used together with *with* or finally.
type	Used to declare a class, record, structure, discriminated union, enumeration type, unit of measure, or type abbreviation.
upcast	Used to convert to a type that is higher in the inheritance chain.
use	Used instead of let for values that require Dispose to be called to free resources.
use!	Used instead of let! in asynchronous workflows and other computation expressions for values that require Dispose to be called to free resources.
val	Used in a signature to indicate a value, or in a type to declare a member, in limited situations.
void	Indicates the .NET void type. Used when interoperating with other .NET languages.
when	Used for Boolean conditions *(when guards)* on pattern matches and to introduce a constraint clause for a generic type parameter.
while	Introduces a looping construct.
with	Used together with the match keyword in pattern matching expressions. Also used in object expressions, record copying expressions, and type extensions to introduce member definitions, and to introduce exception handlers.
yield	Used in a sequence expression to produce a value for a sequence.
yield!	Used in a computation expression to append the result of a given computation expression to a collection of results for the containing computation expression.

Some reserved keywords came from the OCaml language −

asr land lor lsl lsr lxor mod sig

Some other reserved keywords are kept for future expansion of F#.

atomic	break	checked	component	const	constraint	constructor
continue	eager	event	external	fixed	functor	include
method	mixin	object	parallel	process	protected	pure
sealed	tailcall	trait	virtual	volatile		

Comments in F#

F# provides two types of comments −

- One line comment starts with // symbol.
- Multi line comment starts with (* and ends with *).

A Basic Program and Application Entry Point in F#

Generally, you don’t have any explicit entry point for F# programs. When you compile an F#

application, the last file provided to the compiler becomes the entry point and all top level statements in that file are executed from top to bottom.

A well-written program should have a single top-level statement that would call the main loop of the program.

A very minimalistic F# program that would display 'Hello World' on the screen −

```
(* This is a comment *)
(* Sample Hello World program using F# *)
printfn "Hello World!"
```

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

```
Hello World!
```

F# - DATA TYPES

The data types in F# can be classified as follows −

- Integral types
- Floating point types
- Text types
- Other types

Integral Data Type

The following table provides the integral data types of F#. These are basically integer data types.

F# Type	Size	Range	Example	Remarks
sbyte	1 byte	-128 to 127	42y -11y	8-bit signed integer
byte	1 byte	0 to 255	42uy 200uy	8-bit unsigned integer
int16	2 bytes	-32768 to 32767	42s -11s	16-bit signed integer
uint16	2 bytes	0 to 65,535	42us 200us	16-bit unsigned integer
int/int32	4 bytes	-2,147,483,648 to 2,147,483,647	42 -11	32-bit signed integer
uint32	4 bytes	0 to 4,294,967,295	42u	32-bit unsigned

			200u	integer
int64	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	42L -11L	64-bit signed integer
uint64	8 bytes	0 to 18,446,744,073,709,551,615	42UL 200UL	64-bit unsigned integer
bigint	At least 4 bytes	any integer	42I 1499999 9999999 9999999 9999999 9999I	arbitrary precision integer

Example

```
(* single byte integer *)
let x = 268.97f
let y = 312.58f
let z = x + y

printfn "x: %f" x
printfn "y: %f" y
printfn "z: %f" z

(* unsigned 8-bit natural number *)

let p = 2uy
let q = 4uy
let r = p + q

printfn "p: %i" p
printfn "q: %i" q
printfn "r: %i" r

(* signed 16-bit integer *)

let a = 12s
let b = 24s
let c = a + b

printfn "a: %i" a
printfn "b: %i" b
printfn "c: %i" c

(* signed 32-bit integer *)

let d = 212l
let e = 504l
let f = d + e

printfn "d: %i" d
```

```
printfn "e: %i" e
printfn "f: %i" f
```

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

```
x: 1
y: 2
z: 3
p: 2
q: 4
r: 6
a: 12
b: 24
c: 36
d: 212
e: 504
f: 716
```

Floating Point Data Types

The following table provides the floating point data types of F#.

F# Type	Size	Range	Example	Remarks
float32	4 bytes	±1.5e-45 to ±3.4e38	42.0F -11.0F	32-bit signed floating point number (7 significant digits)
float	8 bytes	±5.0e-324 to ±1.7e308	42.0 -11.0	64-bit signed floating point number (15-16 significant digits)
decimal	16 bytes	±1.0e-28 to ±7.9e28	42.0M -11.0M	128-bit signed floating point number (28-29 significant digits)
BigRational	At least 4 bytes	Any rational number.	42N -11N	Arbitrary precision rational number. Using this type requires a reference to FSharp.PowerPack.dll.

Example

```
(* 32-bit signed floating point number *)
(* 7 significant digits *)

let d = 212.098f
let e = 504.768f
let f = d + e

printfn "d: %f" d
printfn "e: %f" e
printfn "f: %f" f

(* 64-bit signed floating point number *)
(* 15-16 significant digits *)
let x = 21290.098
let y = 50446.768
```

```
let z = x + y

printfn "x: %g" x
printfn "y: %g" y
printfn "z: %g" z
```

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

```
d: 212.098000
e: 504.768000
f: 716.866000
x: 21290.1
y: 50446.8
z: 71736.9
```

Text Data Types

The following table provides the text data types of F#.

F# Type	Size	Range	Example	Remarks
char	2 bytes	U+0000 to U+ffff	'x' '\t'	Single unicode characters
string	20 + (2 * string's length) bytes	0 to about 2 billion characters	"Hello" "World"	Unicode text

Example

```
let choice = 'y'
let name = "Zara Ali"
let org = "Tutorials Point"

printfn "Choice: %c" choice
printfn "Name: %s" name
printfn "Organisation: %s" org
```

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

```
Choice: y
Name: Zara Ali
Organisation: Tutorials Point
```

Other Data Types

The following table provides some other data types of F#.

F# Type	Size	Range	Example	Remarks
bool	1 byte	Only two possible values, true or false	true false	Stores boolean values

Example

```
let trueVal = true
let falseVal = false

printfn "True Value: %b" (trueVal)
printfn "False Value: %b" (falseVal)
```

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

```
True Value: true
False Value: false
```

F# - VARIABLES

A variable is a name given to a storage area that our programs can manipulate. Each variable has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Variable Declaration in F#

The **let** keyword is used for variable declaration −

For example,

```
let x = 10
```

It declares a variable x and assigns the value 10 to it.

You can also assign an expression to a variable −

```
let x = 10
let y = 20
let z = x + y
```

The following example illustrates the concept −

Example

```
let x = 10
let y = 20
let z = x + y

printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z
```

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

```
x: 10
y: 20
z: 30
```

Variables in F# are **immutable,** which means once a variable is bound to a value, it can't be changed. They are actually compiled as static read-only properties.

The following example demonstrates this.

Example

```
let x = 10
let y = 20
```

```
let z = x + y

printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z

let x = 15
let y = 20
let z = x + y

printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z
```

When you compile and execute the program, it shows the following error message −

```
Duplicate definition of value 'x'
Duplicate definition of value 'Y'
Duplicate definition of value 'Z'
```

Variable Definition With Type Declaration

A variable definition tells the compiler where and how much storage for the variable should be created. A variable definition may specify a data type and contains a list of one or more variables of that type as shown in the following example.

Example

```
let x:int32 = 10
let y:int32 = 20
let z:int32 = x + y

printfn "x: %d" x
printfn "y: %d" y
printfn "z: %d" z

let p:float = 15.99
let q:float = 20.78
let r:float = p + q

printfn "p: %g" p
printfn "q: %g" q
printfn "r: %g" r
```

When you compile and execute the program, it shows the following error message −

```
x: 10
y: 20
z: 30
p: 15.99
q: 20.78
r: 36.77
```

Mutable Variables

At times you need to change the values stored in a variable. To specify that there could be a change in the value of a declared and assigned variable, in later part of a program, F# provides the **mutable** keyword. You can declare and assign mutable variables using this keyword, whose values you will change.

The **mutable** keyword allows you to declare and assign values in a mutable variable.

You can assign some initial value to a mutable variable using the **let** keyword. However, to assign new subsequent value to it, you need to use the ← operator.

For example,

```
let mutable x = 10
x ← 15
```

The following example will clear the concept −

Example

```
let mutable x = 10
let y = 20
let mutable z = x + y

printfn "Original Values:"
printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z

printfn "Let us change the value of x"
printfn "Value of z will change too."

x <- 15
z <- x + y

printfn "New Values:"
printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z
```

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

```
Original Values:
x: 10
y: 20
z: 30
Let us change the value of x
Value of z will change too.
New Values:
x: 15
y: 20
z: 35
```

F# - OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. F# is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Comparison Operators
- Boolean Operators
- Bitwise Operators

Arithmetic Operators

The following table shows all the arithmetic operators supported by F# language. Assume variable A holds 10 and variable B holds 20 then −

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10

*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
**	Exponentiation Operator, raises an operand to the power of another	B**A will give 20^{10}

Example

```
let a : int32 = 21
let b : int32 = 10

let mutable c = a + b
printfn "Line 1 - Value of c is %d" c

c <- a - b;
printfn "Line 2 - Value of c is %d" c

c <- a * b;
printfn "Line 3 - Value of c is %d" c

c <- a / b;
printfn "Line 4 - Value of c is %d" c

c <- a % b;
printfn "Line 5 - Value of c is %d" c
```

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

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
```

Comparison Operators

The following table shows all the comparison operators supported by F# language. These binary comparison operators are available for integral and floating-point types. These operators return values of type bool.

Assume variable A holds 10 and variable B holds 20, then −

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A <> B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to	(A <= B) is true.

	the value of right operand, if yes then condition becomes true.

Example

```
let mutable a : int32 = 21
let mutable b : int32 = 10

if (a = b) then
   printfn "Line 1 - a is equal to b"
else
   printfn "Line 1 - a is not equal to b"

if (a < b) then
   printfn "Line 2 - a is less than b"
else
   printfn "Line 2 - a is not less than b"

if (a > b) then
   printfn "Line 3 - a is greater than b"
else
   printfn "Line 3 - a is not greater than b"

(* Lets change value of a and b *)
a <- 5
b <- 20

if (a <= b) then
   printfn "Line 4 - a is either less than or equal to b"
else
   printfn "Line4 - a is a is greater than b"
```

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

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
```

Boolean Operators

The following table shows all the Boolean operators supported by F# language. Assume variable A holds **true** and variable B holds **false,** then −

Operator	Description	Example
&&	Called Boolean AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Boolean OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
not	Called Boolean NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not (A && B) is true.

Example

```
let mutable a : bool = true;
let mutable b : bool = true;

if ( a && b ) then
```

```
   printfn "Line 1 - Condition is true"
else
   printfn "Line 1 - Condition is not true"

if ( a || b ) then
   printfn "Line 2 - Condition is true"
else
   printfn "Line 2 - Condition is not true"

(* lets change the value of a *)

a <- false
if ( a && b ) then
   printfn "Line 3 - Condition is true"
else
   printfn "Line 3 - Condition is not true"

if ( a || b ) then
   printfn "Line 4 - Condition is true"
else
   printfn "Line 4 - Condition is not true"
```

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

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation. The truth tables for &&& (bitwise AND), ||| (bitwise OR), and ^^^ (bitwise exclusive OR) are as follows −

p	q	p &&& q	p ||| q	p ^^^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

A&&&B = 0000 1100

A|||B = 0011 1101

A^^^B = 0011 0001

~~~A = 1100 0011

The Bitwise operators supported by F# language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

Operator	Description	Example
~~~

&&&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A &&& B) will give 12, which is 0000 1100
|||	Binary OR Operator copies a bit if it exists in either operand.	(A ||| B) will give 61, which is 0011 1101
^^^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^^^ B) will give 49, which is 0011 0001
~~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~~~A) will give -61, which is 1100 0011 in 2's complement form.
<<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A <<< 2 will give 240 which is 1111 0000
>>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >>> 2 will give 15 which is 0000 1111

Example

```
let a : int32 = 60 // 60 = 0011 1100
let b : int32 = 13 // 13 = 0000 1101
let mutable c : int32 = 0

c <- a &&& b // 12 = 0000 1100
printfn "Line 1 - Value of c is %d" c

c <- a ||| b // 61 = 0011 1101
printfn "Line 2 - Value of c is %d" c

c <- a ^^^ b // 49 = 0011 0001
printfn "Line 3 - Value of c is %d" c

c = ~~~a // -61 = 1100 0011
printfn "Line 4 - Value of c is %d" c

c <- a <<< 2 // 240 = 1111 0000
printfn "Line 5 - Value of c is %d" c

c <- a >>> 2 // 15 = 0000 1111
printfn "Line 6 - Value of c is %d" c
```

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

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is 49
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Operators Precedence

The following table shows the order of precedence of operators and other expression keywords in the F# language, from lowest precedence to the highest precedence.

Operator	Associativity
as	Right

when	Right
| (pipe)	Left
;	Right
let	Non associative
function, fun, match, try	Non associative
if	Non associative
→	Right
:=	Right
,	Non associative
or, ||	Left
&, &&	Left
< op, >op, =, |op, &op	Left
&&& , |||, ^^^, ~~~, <<<, >>>	Left
^ op	Right
::	Right
:?>, :?	Non associative
- op, +op, (binary)	Left
* op, /op, %op	Left
** op	Right
f x (function application)	Left
| (pattern match)	Right
prefix operators (+op, -op, %, %%, &, &&, !op, ~op)	Left
.	Left
f(x)	Left
f<types>	Left

Example

```
let a : int32 = 20
let b : int32 = 10
let c : int32 = 15
let d : int32 = 5

let mutable e : int32 = 0
e <- (a + b) * c / d // ( 30 * 15 ) / 5
printfn "Value of (a + b) * c / d is : %d" e

e <- ((a + b) * c) / d // (30 * 15 ) / 5
printfn "Value of ((a + b) * c) / d is : %d" e

e <- (a + b) * (c / d) // (30) * (15/5)
printfn "Value of (a + b) * (c / d) is : %d" e
```

```
e <- a + (b * c) / d // 20 + (150/5)
printfn "Value of a + (b * c) / d is : %d" e
```

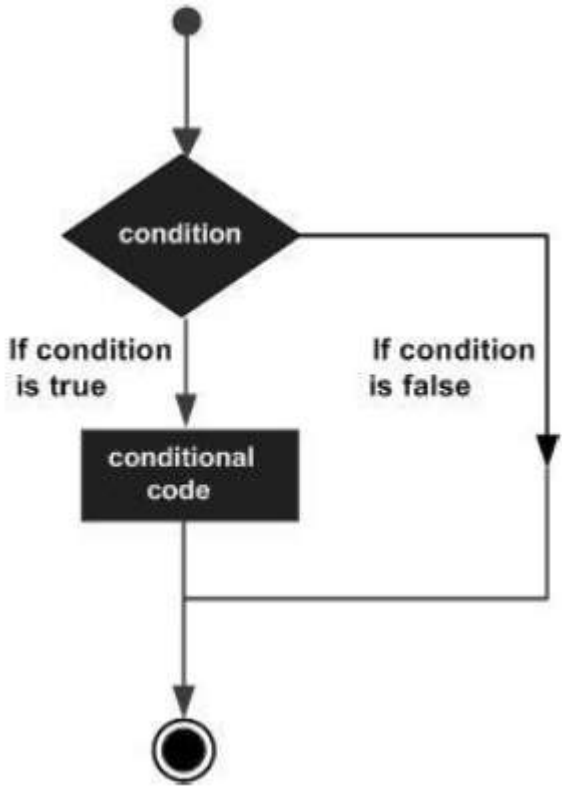
When you compile and execute the program, it yields the following output −

```
Value of (a + b) * c / d is : 90
Value of ((a + b) * c) / d is : 90
Value of (a + b) * (c / d) is : 90
Value of a + (b * c) / d is : 50
```

F# - DECISION MAKING

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program. It should be along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



F# programming language provides the following types of decision making statements.

Statement	Description
if /then statement	An **if/then statement** consists of a Boolean expression followed by one or more statements.
if/then/ else statement	An **if/then statement** can be followed by an optional **else statement,** which executes when the Boolean expression is false.
if/then/elif/else statement	An **if/then/elif/else** statement allows you to have multiple else branches.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

F#-if/then Statement

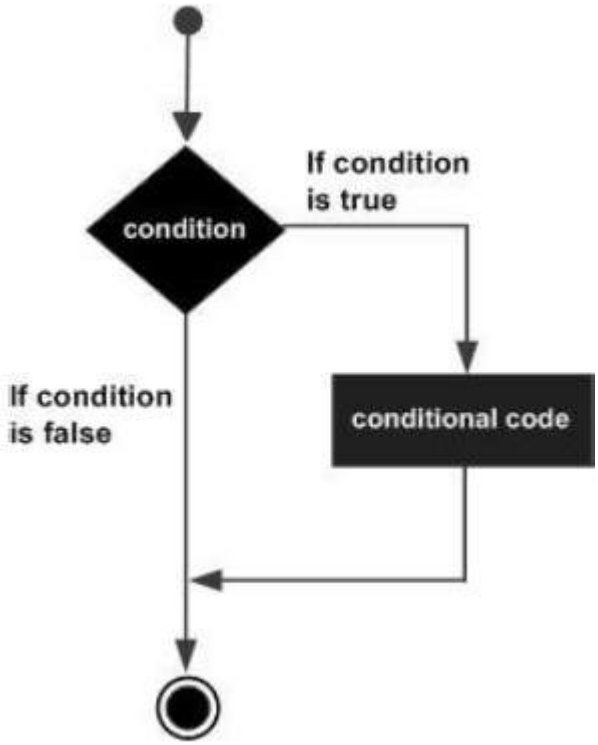
An if/then statement consists of a Boolean expression followed by one or more statements.

Syntax

The if/then construct in F# has the following syntax −

```
(* simple if *)
if expr then
   expr
```

Flow diagram



Example

```
let a : int32 = 10

(* check the boolean condition using if statement *)
if (a < 20) then
   printfn "a is less than 20\n"
   printfn "Value of a is: %d" a
```

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

```
a is less than 20

Value of a is: 10
```

F#-if/then/else Statement

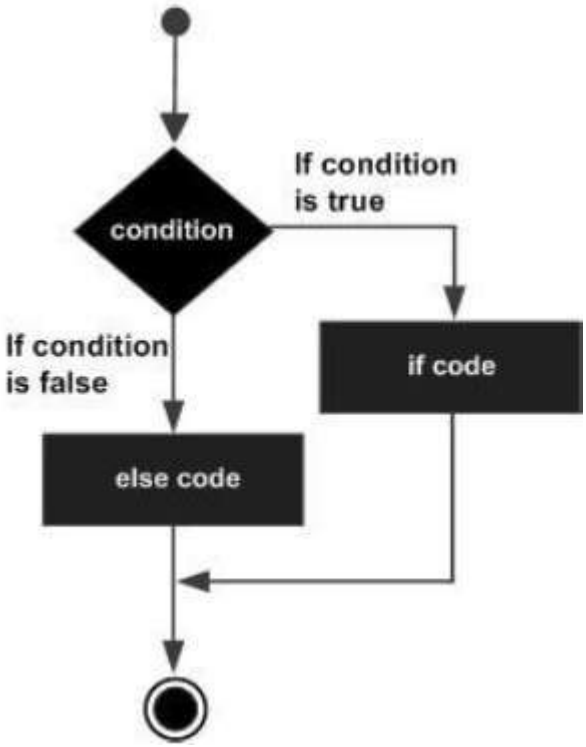
An **if/then** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax

The syntax of an if/then/else statement in F# programming language is −

```
if expr then
   expr
else
   expr
```

Flow Diagram



Example

```
let a : int32 = 100

(* check the boolean condition using if statement *)

if (a < 20) then
   printfn "a is less than 20\n"
else
   printfn "a is not less than 20\n"
   printfn "Value of a is: %d" a
```

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

```
a is not less than 20

Value of a is: 100
```

F#-if/then/elif/else Statement

An **if/then/elif/else** construct has multiple else branches.

Syntax

The syntax of an if/then/elif/else statement in F# programming language is −

```
if expr then
   expr
elif expr then
   expr
elif expr then
   expr
```

```
...
else
   expr
```

Example

```
let a : int32 = 100

(* check the boolean condition using if statement *)

if (a = 10) then
   printfn "Value of a is 10\n"
elif (a = 20) then
   printfn " Value of a is 20\n"
elif (a = 30) then
   printfn " Value of a is 30\n"
else
   printfn " None of the values are matching\n"
   printfn "Value of a is: %d" a
```

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

```
None of the values are matching

Value of a is: 100
```

F#-Nested if Statements

It is always legal in F# programming to nest if/then or if/then/else statements, which means you can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Syntax

```
if expr then
   expr
   if expr then
      expr
   else
      expr
else
   expr
```

Example

```
let a : int32 = 100
let b : int32 = 200

(* check the boolean condition using if statement *)

if (a = 100) then
(* if condition is true then check the following *)

   if (b = 200) then
      printfn "Value of a is 100 and b is 200\n"
printfn "Exact value of a is: %d" a
printfn "Exact value of b is: %d" b
```

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

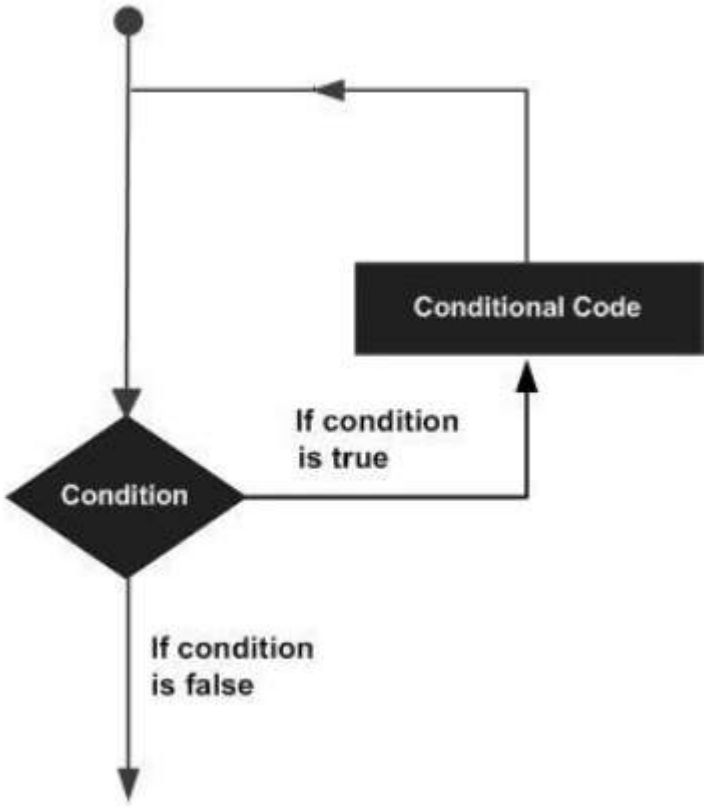
```
Value of a is 100 and b is 200

Exact value of a is: 100
Exact value of b is: 200
```

F# - LOOPS

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



F# provides the following types of loops to handle the looping requirements.

Loop Type	Description
for… to and for… downto expressions	The for...to expression is used to iterate in a loop over a range of values of a loop variable. The for… downto expression reduces the value of loop variable.
for … in expression	This form of for loop is used to iterate over collections of items i.e., loops over collections and sequences
While…do loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
nested loops	You can use one or more loop inside any other for or while loop.

F#-for...to and for...downto Expressions

A **for loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for...to** loop in F# programming language is −

```
for var = start-expr to end-expr do
   ... // loop body
```

The syntax of a for...downto loop in F# programming language is −

```
for var = start-expr downto end-expr do
   ... // loop body
```

Example 1

The following program prints out the numbers 1 - 20 −

```
let main() =
   for i = 1 to 20 do
      printfn "i: %i" i
main()
```

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

```
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
i: 11
i: 12
i: 13
i: 14
i: 15
i: 16
i: 17
i: 18
i: 19
i: 20
```

Example 2

The following program counts in reverse and prints out the numbers 20 - 1 −

```
let main() =
   for i = 20 downto 1 do
      printfn "i: %i" i
main()
```

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

```
i: 20
i: 19
i: 18
i: 17
i: 16
i: 15
i: 14
i: 13
i: 12
i: 11
i: 10
```

```
i: 9
i: 8
i: 7
i: 6
i: 5
i: 4
i: 3
i: 2
i: 1
```

F#-for...in Expressions

This looping construct is used to iterate over the matches of a pattern in an enumerable collection such as a range expression, sequence, list, array, or other construct that supports enumeration.

Syntax

```
for pattern in enumerable-expression do
   body-expression
```

Example

The following program illustrates the concept −

```
// Looping over a list.
let list1 = [ 10; 25; 34; 45; 78 ]
for i in list1 do
   printfn "%d" i

// Looping over a sequence.
let seq1 = seq { for i in 1 .. 10 -> (i, i*i) }
for (a, asqr) in seq1 do
   printfn "%d squared is %d" a asqr
```

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

```
10
25
34
45
78
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

F#-While...do Expressions

The **while...do** expression is used to perform iterative execution while a specified test condition is true.

Syntax

```
while test-expression do
   body-expression
```

The test-expression is evaluated first; if it is true, the body-expression is executed and the test expression is evaluated again. The body-expression must have type unit, i.e., it should not return

any value. If the test expression is false, the iteration ends.

Example

```
let mutable a = 10
while (a < 20) do
   printfn "value of a: %d" a
   a <- a + 1
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

F#-Nested Loops

F# programming language allows to use one loop inside another loop.

Syntax

The syntax for a nested for loop statement could be as follows −

```
for var1 = start-expr1 to end-expr1 do
   for var2 = start-expr2 to end-expr2 do
      ... // loop body
```

The syntax for a nested while loop statement could be as follows −

```
while test-expression1 do
   while test-expression2 do
      body-expression
```

Example

```
let main() =
   for i = 1 to 5 do
      printf "\n"
      for j = 1 to 3 do
         printf "*"
main()
```

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

```
***
***
***
***
***
```

F# - FUNCTIONS

In F#, functions work like data types. You can declare and use a function in the same way like any other variable.

Since functions can be used like any other variables, you can −

- Create a function, with a name and associate that name with a type.
- Assign it a value.
- Perform some calculation on that value.
- Pass it as a parameter to another function or sub-routine.
- Return a function as the result of another function.

Defining a Function

Functions are defined by using the **let** keyword. A function definition has the following syntax −

```
let [inline] function-name parameter-list [ : return-type ]
= function-body
```

Where,

- **function-name** is an identifier that represents the function.
- **parameter-list** gives the list of parameters separated by spaces. You can also specify an explicit type for each parameter and if not specified compiler tends to deduce it from the function body (like variables).
- **function-body** consists of an expression, or a compound expression consisting of a number of expressions. The final expression in the function body is the return value.
- **return-type** is a colon followed by a type and is optional. If the return type is not specified, then the compiler determines it from the final expression in the function body.

Parameters of a Function

You list the names of parameters right after the function name. You can specify the type of a parameter. The type of the parameter should follow the name of the parameter separated by a colon.

If no parameter type is specified, it is inferred by the compiler.

For example −

```
let doubleIt (x : int) = 2 * x
```

Calling a Function

A function is called by specifying the function name followed by a space and then any arguments separated by spaces.

For example −

```
let vol = cylinderVolume 3.0 5.0
```

The following programs illustrate the concepts.

Example 1

The following program calculates the volume of a cylinder when the radius and length are given as parameters.

```
// the function calculates the volume of
// a cylinder with radius and length as parameters

let cylinderVolume radius length : float =

   // function body
```

```
   let pi = 3.14159
   length * pi * radius * radius

let vol = cylinderVolume 3.0 5.0
printfn " Volume: %g " vol
```

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

```
Volume: 141.372
```

Example 2

The following program returns the larger value of two given parameters −

```
// the function returns the larger value between two
// arguments

let max num1 num2 : int32 =
   // function body
   if(num1>num2)then
      num1
   else
      num2

let res = max 39 52
printfn " Max Value: %d " res
```

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

```
Max Value: 52
```

Example 3

```
let doubleIt (x : int) = 2 * x
printfn "Double 19: %d" ( doubleIt(19))
```

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

```
Double 19: 38
```

Recursive Functions

Recursive functions are functions that call themselves.

You define a recursive using the **let rec** keyword combination.

Syntax for defining a recursive function is −

```
//Recursive function definition
let rec function-name parameter-list = recursive-function-body
```

For example −

```
let rec fib n = if n < 2 then 1 else fib (n - 1) &plus; fib (n - 2)
```

Example 1

The following program returns Fibonacci 1 to 10 −

```
let rec fib n = if n < 2 then 1 else fib (n - 1) &plus; fib (n - 2)
for i = 1 to 10 do
   printfn "Fibonacci %d: %d" i (fib i)
```

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

```
Fibonacci 1: 1
Fibonacci 2: 2
Fibonacci 3: 3
Fibonacci 4: 5
Fibonacci 5: 8
Fibonacci 6: 13
Fibonacci 7: 21
Fibonacci 8: 34
Fibonacci 9: 55
Fibonacci 10: 89
```

Example 2

The following program returns factorial 8 −

```
open System
let rec fact x =
   if x < 1 then 1
   else x * fact (x - 1)
Console.WriteLine(fact 8)
```

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

```
40320
```

Arrow Notations in F#

F# reports about data type in functions and values, using a chained arrow notation. Let us take an example of a function that takes one *int* input, and returns a string. In arrow notation, it is written as −

```
int -> string
```

Data types are read from left to right.

Let us take another hypothetical function that takes two int data inputs and returns a string.

```
let mydivfunction x y = (x / y).ToString();;
```

F# reports the data type using chained arrow notation as −

```
val mydivfunction : x:int -> y:int -> string
```

The return type is represented by the rightmost data type in chained arrow notation.

Some more examples −

Notation	Meaning
float → float → float	The function takes two *float* inputs, returns another *float*.
int → string → float	The function takes an *int* and a *string* input, returns a *float*.

Lambda Expressions

A **lambda expression** is an unnamed function.

Let us take an example of two functions −

```
let applyFunction ( f: int -> int -> int) x y = f x y
let mul x y = x * y
let res = applyFunction mul 5 7
printfn "%d" res
```

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

```
35
```

Now in the above example, if instead of defining the function *mul,* we could have used lambda expressions as −

```
let applyFunction ( f: int -> int -> int) x y = f x y
let res = applyFunction (fun x y -> x * y ) 5 7
printfn "%d" res
```

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

```
35
```

Function Composition and Pipelining

In F#, one function can be composed from other functions.

The following example shows the composition of a function named f, from two functions function1 and function2 −

```
let function1 x = x + 1
let function2 x = x * 5

let f = function1 >> function2
let res = f 10
printfn "%d" res
```

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

```
55
```

F# also provides a feature called **pipelining of functions.** Pipelining allows function calls to be chained together as successive operations.

The following example shows that −

```
let function1 x = x + 1
let function2 x = x * 5

let res = 10 |> function1 |> function2
printfn "%d" res
```

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

```
55
```

F# - STRINGS

In F#, the string type represents immutable text as a sequence of Unicode characters.

String Literals

String literals are delimited by the quotation mark (") character.

Some special characters are there for special uses like newline, tab, etc. They are encoded using

backslash (\) character. The backslash character and the related character make the escape sequence. The following table shows the escape sequence supported by F#.

Character	Escape sequence
Backspace	\b
Newline	\n
Carriage return	\r
Tab	\t
Backslash	\\
Quotation mark	\"
Apostrophe	\'
Unicode character	\uXXXX or \UXXXXXXXX (where X indicates a hexadecimal digit)

Ways of Ignoring the Escape Sequence

The following two ways makes the compiler ignore the escape sequence −

- Using the @ symbol.
- Enclosing the string in triple quotes.

When a string literal is preceded by the @ symbol, it is called a **verbatim string.** In that way, all escape sequences in the string are ignored, except that two quotation mark characters are interpreted as one quotation mark character.

When a string is enclosed by triple quotes, then also all escape sequences are ignored, including double quotation mark characters.

Example

The following example demonstrates this technique showing how to work with XML or other structures that include embedded quotation marks −

```
// Using a verbatim string
let xmldata = @"<book author=""Lewis, C.S"" title=""Narnia"">"
printfn "%s" xmldata
```

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

```
<book author="Lewis, C.S" title="Narnia">
```

Basic Operators on Strings

The following table shows the basic operations on strings −

Value	Description
collect : (char → string) → string → string	Creates a new string whose characters are the results of applying a specified function to each of the characters of the input string and concatenating the resulting strings.
concat : string → seq<string> → string	Returns a new string made by concatenating the given strings with a separator.

exists : (char → bool) → string → bool	Tests if any character of the string satisfies the given predicate.
forall : (char → bool) → string → bool	Tests if all characters in the string satisfy the given predicate.
init : int → (int → string) → string	Creates a new string whose characters are the results of applying a specified function to each index and concatenating the resulting strings.
iter : (char → unit) → string → unit	Applies a specified function to each character in the string.
iteri : (int → char → unit) → string → unit	Applies a specified function to the index of each character in the string and the character itself.
length : string → int	Returns the length of the string.
map : (char → char) → string → string	Creates a new string whose characters are the results of applying a specified function to each of the characters of the input string.
mapi : (int → char → char) → string → string	Creates a new string whose characters are the results of applying a specified function to each character and index of the input string.
replicate : int → string → string	Returns a string by concatenating a specified number of instances of a string.

The following examples demonstrate the uses of some of the above functionalities −

Example 1

The String.collect function builds a new string whose characters are the results of applying a specified function to each of the characters of the input string and concatenating the resulting strings.

```
let collectTesting inputS =
   String.collect (fun c -> sprintf "%c " c) inputS
printfn "%s" (collectTesting "Happy New Year!")
```

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

```
H a p p y N e w Y e a r !
```

Example 2

The String.concat function concatenates a given sequence of strings with a separator and returns a new string.

```
let strings = [ "Tutorials Point"; "Coding Ground"; "Absolute Classes" ]
let ourProducts = String.concat "\n" strings
printfn "%s" ourProducts
```

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

```
Tutorials Point
Coding Ground
Absolute Classes
```

Example 3

The String.replicate method returns a string by concatenating a specified number of instances of a string.

```
printfn "%s" <| String.replicate 10 "*! "
```

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

```
*! *! *! *! *! *! *! *! *! *!
```

F# - OPTIONS

The **option** type in F# is used in calculations when there may or may not exist a value for a variable or function. Option types are used for representing optional values in calculations. They can have two possible values − **Some(x)** or **None**.

For example, a function performing a division will return a value in normal situation, but will throw exceptions in case of a zero denominator. Using options here will help to indicate whether the function has succeeded or failed.

An option has an underlying type and can hold a value of that type, or it might not have a value.

Using Options

Let us take the example of division function. The following program explains this −

Let us write a function div, and send two arguments to it 20 and 5 −

```
let div x y = x / y
let res = div 20 5
printfn "Result: %d" res
```

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

```
Result: 4
```

If the second argument is zero, then the program throws an exception −

```
let div x y = x / y
let res = div 20 0
printfn "Result: %d" res
```

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

```
Unhandled Exception:
System.DivideByZeroException: Division by zero
```

In such cases, we can use option types to return Some (value) when the operation is successful or None if the operation fails.

The following example demonstrates the use of options −

Example

```
let div x y =
   match y with
   | 0 -> None
   | _ -> Some(x/y)

let res : int option = div 20 4
printfn "Result: %A " res
```

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

```
Result: Some 5
```

Option Properties and Methods

The option type supports the following properties and methods −

Property or method	Type	Description
None	'T option	A static property that enables you to create an option value that has the **None value**.
IsNone	bool	Returns **true** if the option has the **None** value.
IsSome	bool	Returns **true** if the option has a value that is not **None**.
Some	'T option	A static member that creates an option that has a value that is not **None**.
Value	'T	Returns the underlying value, or throws a NullReferenceException if the value is **None**.

Example 1

```
let checkPositive (a : int) =
   if a > 0 then
      Some(a)
   else
      None

let res : int option = checkPositive(-31)
printfn "Result: %A " res
```

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

```
Result: <null>
```

Example 2

```
let div x y =
   match y with
   | 0 -> None
   | _ -> Some(x/y)

let res : int option = div 20 4
printfn "Result: %A " res
printfn "Result: %A " res.Value
```

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

```
Result: Some 5
Result: 5
```

Example 3

```
let isHundred = function
   | Some(100) -> true
   | Some(_) | None -> false

printfn "%A" (isHundred (Some(45)))
printfn "%A" (isHundred (Some(100)))
```

```
printfn "%A" (isHundred None)
```

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

```
false
true
false
```

F# - TUPLES

A **tuple** is a comma-separated collection of values. These are used for creating ad hoc data structures, which group together related values.

For example, (“Zara Ali”, “Hyderabad”, 10) is a 3-tuple with two string values and an int value, it has the type (string * string * int).

Tuples could be pairs, triples, and so on, of the same or different types.

Some examples are provided here −

```
// Tuple of two integers.
( 4, 5 )

// Triple of strings.
( "one", "two", "three" )

// Tuple of unknown types.
( a, b )

// Tuple that has mixed types.
( "Absolute Classes", 1, 2.0 )

// Tuple of integer expressions.
( a * 4, b + 7)
```

Example

This program has a function that takes a tuple of four float values and returns the average −

```
let averageFour (a, b, c, d) =
   let sum = a + b + c + d
   sum / 4.0

let avg:float = averageFour (4.0, 5.1, 8.0, 12.0)
printfn "Avg of four numbers: %f" avg
```

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

```
Avg of four numbers: 7.275000
```

Accessing Individual Tuple Members

The individual members of a tuple could be assessed and printed using pattern matching.

The following example illustrates the concept −

Example

```
let display tuple1 =
   match tuple1 with
   | (a, b, c) -> printfn "Detail Info: %A %A %A" a b c

display ("Zara Ali", "Hyderabad", 10 )
```

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

```
Detail Info: "Zara Ali" "Hyderabad" 10
```

F# has two built-in functions, **fst** and **snd,** which return the first and second items in a 2-tuple.

The following example illustrates the concept −

Example

```
printfn "First member: %A" (fst(23, 30))
printfn "Second member: %A" (snd(23, 30))

printfn "First member: %A" (fst("Hello", "World!"))
printfn "Second member: %A" (snd("Hello", "World!"))

let nameTuple = ("Zara", "Ali")

printfn "First Name: %A" (fst nameTuple)
printfn "Second Name: %A" (snd nameTuple)
```

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

```
First member: 23
Second member: 30
First member: "Hello"
Second member: "World!"
First Name: "Zara"
Second Name: "Ali"
```

F# - RECORDS

A **record** is similar to a tuple, however it contains named fields. For example,

```
type website =
   { title : string;
      url : string }
```

Defining Record

A record is defined as a type using the **type** keyword, and the fields of the record are defined as a semicolon-separated list.

Syntax for defining a record is −

```
type recordName =
   { [ fieldName : dataType ] + }
```

Creating a Record

You can create a record by specifying the record's fields. For example, let us create a *website* record named *homepage* −

```
let homepage = { Title = "TutorialsPoint"; Url = "www.tutorialspoint.com" }
```

The following examples will explain the concepts −

Example 1

This program defines a record type named website. Then it creates some records of type website and prints the records.

```
(* defining a record type named website *)
```

```
type website =
   { Title : string;
      Url : string }

(* creating some records *)
let homepage = { Title = "TutorialsPoint"; Url = "www.tutorialspoint.com" }
let cpage = { Title = "Learn C"; Url = "www.tutorialspoint.com/cprogramming/index.htm" }
let fsharppage = { Title = "Learn F#"; Url = "www.tutorialspoint.com/fsharp/index.htm" }
let csharppage = { Title = "Learn C#"; Url = "www.tutorialspoint.com/csharp/index.htm" }

(*printing records *)
(printfn "Home Page: Title: %A \n \t URL: %A") homepage.Title homepage.Url
(printfn "C Page: Title: %A \n \t URL: %A") cpage.Title cpage.Url
(printfn "F# Page: Title: %A \n \t URL: %A") fsharppage.Title fsharppage.Url
(printfn "C# Page: Title: %A \n \t URL: %A") csharppage.Title csharppage.Url
```

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

```
Home Page: Title: "TutorialsPoint"
       URL: "www.tutorialspoint.com"
C Page: Title: "Learn C"
      URL: "www.tutorialspoint.com/cprogramming/index.htm"
F# Page: Title: "Learn F#"
      URL: "www.tutorialspoint.com/fsharp/index.htm"
C# Page: Title: "Learn C#"
      URL: "www.tutorialspoint.com/csharp/index.htm"
```

Example 2

```
type student =
   { Name : string;
      ID : int;
      RegistrationText : string;
      IsRegistered : bool }

let getStudent name id =
   { Name = name; ID = id; RegistrationText = null; IsRegistered = false }

let registerStudent st =
   { st with
      RegistrationText = "Registered";
      IsRegistered = true }

let printStudent msg st =
   printfn "%s: %A" msg st

let main() =
   let preRegisteredStudent = getStudent "Zara" 10
   let postRegisteredStudent = registerStudent preRegisteredStudent

   printStudent "Before Registration: " preRegisteredStudent
   printStudent "After Registration: " postRegisteredStudent

main()
```

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

```
Before Registration: : {Name = "Zara";
   ID = 10;
   RegistrationText = null;
   IsRegistered = false;}
After Registration: : {Name = "Zara";
   ID = 10;
   RegistrationText = "Registered";
   IsRegistered = true;}
```

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 → ^U) → 'T list → ^U	Returns the average of the elements generated by applying the function to each element of the list.
choose : ('T → 'U option) → '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 → 'U list) → '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 → bool) → 'T list → bool	Tests if any element of the list satisfies the given predicate.

exists2 : ('T1 → 'T2 → bool) → 'T1 list → 'T2 list → bool	Tests if any pair of corresponding elements of the lists satisfies the given predicate.
filter : ('T → 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 → bool) → 'T list → 'T	Returns the first element for which the given function returns **true**.
findIndex : ('T → bool) → 'T list → int	Returns the index of the first element in the list that satisfies the given predicate.
fold : ('State → 'T → '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 (... (f s i0) i1 ...) iN.
fold2 : ('State → 'T1 → 'T2 → '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 (... (f s i0 j0)...) iN jN.
foldBack : ('T → 'State → '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 (...(f iN s)).
foldBack2 : ('T1 → 'T2 → 'State → '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 (...(f iN jN s)).
forall : ('T → bool) → 'T list → bool	Tests if all elements of the collection satisfy the given predicate.
forall2 : ('T1 → 'T2 → 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 → '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 → unit) → 'T list → unit	Applies the given function to each element of the collection.
iter2 : ('T1 → 'T2 → unit) → 'T1 list → 'T2 list → unit	Applies the given function to two collections simultaneously. The collections must have identical size.

iteri : (int → 'T → 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 → 'T1 → 'T2 → 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 → '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 → 'T2 → '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 → 'T2 → 'T3 → '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 → 'T → '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 → 'T1 → 'T2 → '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 → '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 → '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 → 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 → int) → 'T list → 'T list	Returns a list with all elements permuted according to the specified permutation.
pick : ('T → 'U option) → 'T list → 'U	Applies the given function to successive elements, returning the first result where function returns **Some** for some value.

reduce : ('T → 'T → '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 (... (f i0 i1) i2 ...) iN.
reduceBack : ('T → 'T → '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 (...(f iN-1 iN)).
replicate : (int → 'T → 'T list)	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 → 'T → '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 → 'State → '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 → 'Key) → 'T list → 'T list	Sorts the given list using keys given by the given projection. Keys are compared using Operators.compare.
sortWith : ('T → 'T → 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 → ^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 → 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 → 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 → 'U option) → '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.
```

F# - SEQUENCES

Sequences, like lists also represent an ordered collection of values. However, the elements in a sequence or sequence expression are computed when required. They are not computed at once, and for this reason they are used to represent infinite data structures.

Defining Sequences

Sequences are defined using the following syntax −

```
seq { expr }
```

For example,

```
let seq1 = seq { 1 .. 10 }
```

Creating Sequences and Sequences Expressions

Similar to lists, you can create sequences using ranges and comprehensions.

Sequence expressions are the expressions you can write for creating sequences. These can be done −

- By specifying the range.
- By specifying the range with increment or decrement.
- By using the **yield** keyword to produce values that become part of the sequence.
- By using the → operator.

The following examples demonstrate the concept −

Example 1

```
(* Sequences *)
let seq1 = seq { 1 .. 10 }

(* ascending order and increment*)
printfn "The Sequence: %A" seq1
let seq2 = seq { 1 .. 5 .. 50 }

(* descending order and decrement*)
printfn "The Sequence: %A" seq2
let seq3 = seq {50 .. -5 .. 0}
printfn "The Sequence: %A" seq3

(* using yield *)
let seq4 = seq { for a in 1 .. 10 do yield a, a*a, a*a*a }
printfn "The Sequence: %A" seq4
```

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

```
The Sequence: seq [1; 2; 3; 4; ...]
The Sequence: seq [1; 6; 11; 16; ...]
The Sequence: seq [50; 45; 40; 35; ...]
The Sequence: seq [(1, 1, 1); (2, 4, 8); (3, 9, 27); (4, 16, 64); ...]
```

Example 2

The following program prints the prime numbers from 1 to 50 −

```
(* Recursive isprime function. *)
let isprime n =
   let rec check i =
      i > n/2 || (n % i <> 0 && check (i + 1))
   check 2

let primeIn50 = seq { for n in 1..50 do if isprime n then yield n }
for x in primeIn50 do
   printfn "%d" x
```

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

```
1
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
```

Basic Operations on Sequence

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

Value	Description
append : seq<'T> → seq<'T> → seq<'T>	Wraps the two given enumerations as a single concatenated enumeration.
average : seq<^T> → ^T	Returns the average of the elements in the sequence.
averageBy : ('T → ^U) → seq<'T> → ^U	Returns the average of the results generated by applying the function to each element of the sequence.
cache : seq<'T> → seq<'T>	Returns a sequence that corresponds to a cached version of the input sequence.
cast : IEnumerable → seq<'T>	Wraps a loosely-typed System. Collections sequence as a typed sequence.
choose : ('T → 'U option) → seq<'T> → seq<'U>	Applies the given function to each element of the list. Return the list comprised of the results

	for each element where the function returns **Some**.
collect : ('T → 'Collection) → seq<'T> → seq<'U>	Applies the given function to each element of the sequence and concatenates all the results.
compareWith : ('T → 'T → int) → seq<'T> → seq<'T> → int	Compares two sequences using the given comparison function, element by element.
concat : seq<'Collection> → seq<'T>	Combines the given enumeration-of-enumerations as a single concatenated enumeration.
countBy : ('T → 'Key) → seq<'T> → seq<'Key * int>	Applies a key-generating function to each element of a sequence and return a sequence yielding unique keys and their number of occurrences in the original sequence.
delay : (unit → seq<'T>) → seq<'T>	Returns a sequence that is built from the given delayed specification of a sequence.
distinct : seq<'T> → seq<'T>	Returns a sequence that contains no duplicate entries according to generic hash and equality comparisons on the entries. If an element occurs multiple times in the sequence then the later occurrences are discarded.
distinctBy : ('T → 'Key) → seq<'T> → seq<'T>	Returns a sequence that contains no duplicate entries according to the generic hash and equality comparisons on the keys returned by the given key-generating function. If an element occurs multiple times in the sequence then the later occurrences are discarded.
empty : seq<'T>	Creates an empty sequence.
exactlyOne : seq<'T> → 'T	Returns the only element of the sequence.
exists : ('T → bool) → seq<'T> → bool	Tests if any element of the sequence satisfies the given predicate.
exists2 : ('T1 → 'T2 → bool) → seq<'T1> → seq<'T2> → bool	Tests if any pair of corresponding elements of the input sequences satisfies the given predicate.
filter : ('T → bool) → seq<'T> → seq<'T>	Returns a new collection containing only the elements of the collection for which the given predicate returns **true**.
find : ('T → bool) → seq<'T> → 'T	Returns the first element for which the given function returns **true**.
findIndex : ('T → bool) → seq<'T> → int	Returns the index of the first element for which the given function returns **true**.
fold : ('State → 'T → 'State) → 'State → seq<'T> → 'State	Applies a function to each element of the collection, threading an accumulator argument through the computation. If the input function is f and the elements are i0...iN, then this function computes f (... (f s i0)...) iN.
forall : ('T → bool) → seq<'T> → bool	Tests if all elements of the sequence satisfy the given predicate.
forall2 : ('T1 → 'T2 → bool) → seq<'T1> → seq<'T2> → bool	Tests the all pairs of elements drawn from the two sequences satisfy the given predicate. If one sequence is shorter than the other then the

	remaining elements of the longer sequence are ignored.
groupBy : ('T → 'Key) → seq<'T> → seq<'Key * seq<'T>>	Applies a key-generating function to each element of a sequence and yields a sequence of unique keys. Each unique key has also contains a sequence of all elements that match to this key.
head : seq<'T> → 'T	Returns the first element of the sequence.
init : int → (int → 'T) → seq<'T>	Generates a new sequence which, when iterated, returns successive elements by calling the given function, up to the given count. The results of calling the function are not saved, that is, the function is reapplied as necessary to regenerate the elements. The function is passed the index of the item being generated.
initInfinite : (int → 'T) → seq<'T>	Generates a new sequence which, when iterated, will return successive elements by calling the given function. The results of calling the function are not saved, that is, the function will be reapplied as necessary to regenerate the elements. The function is passed the index of the item being generated.
isEmpty : seq<'T> → bool	Tests whether a sequence has any elements.
iter : ('T → unit) → seq<'T> → unit	Applies the given function to each element of the collection.
iter2 : ('T1 → 'T2 → unit) → seq<'T1> → seq<'T2> → unit	Applies the given function to two collections simultaneously. If one sequence is shorter than the other then the remaining elements of the longer sequence are ignored.
iteri : (int → 'T → unit) → seq<'T> → unit	Applies the given function to each element of the collection. The integer passed to the function indicates the index of element.
last : seq<'T> → 'T	Returns the last element of the sequence.
length : seq<'T> → int	Returns the length of the sequence.
map : ('T → 'U) → seq<'T> → seq<'U>	Creates a new collection whose elements are the results of applying the given function to each of the elements of the collection. The given function will be applied as elements are demanded using the MoveNext method on enumerators retrieved from the object.
map2 : ('T1 → 'T2 → 'U) → seq<'T1> → seq<'T2> → seq<'U>	Creates a new collection whose elements are the results of applying the given function to the corresponding pairs of elements from the two sequences. If one input sequence is shorter than the other then the remaining elements of the longer sequence are ignored.
mapi : (int → 'T → 'U) → seq<'T> → seq<'U>	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.
max : seq<'T> → 'T	Returns the greatest of all elements of the sequence, compared by using Operators.max.

maxBy : ('T → 'U) → seq<'T> → 'T	Returns the greatest of all elements of the sequence, compared by using Operators.max on the function result.
min : seq<'T> → 'T	Returns the lowest of all elements of the sequence, compared by using Operators.min.
minBy : ('T → 'U) → seq<'T> → 'T	Returns the lowest of all elements of the sequence, compared by using Operators.min on the function result.
nth : int → seq<'T> → 'T	Computes the *nth* element in the collection.
ofArray : 'T array → seq<'T>	Views the given array as a sequence.
ofList : 'T list → seq<'T>	Views the given list as a sequence.
pairwise : seq<'T> → seq<'T * 'T>	Returns a sequence of each element in the input sequence and its predecessor, with the exception of the first element which is only returned as the predecessor of the second element.
pick : ('T → 'U option) → seq<'T> → 'U	Applies the given function to successive elements, returning the first value where the function returns a **Some** value.
readonly : seq<'T> → seq<'T>	Creates a new sequence object that delegates to the given sequence object. This ensures the original sequence cannot be rediscovered and mutated by a type cast. For example, if given an array the returned sequence will return the elements of the array, but you cannot cast the returned sequence object to an array.
reduce : ('T → 'T → 'T) → seq<'T> → 'T	Applies a function to each element of the sequence, threading an accumulator argument through the computation. Begin by applying the function to the first two elements. Then feed this result into the function along with the third element and so on. Return the final result.
scan : ('State → 'T → 'State) → 'State → seq<'T> → seq<'State>	Like Seq.fold, but computes on-demand and returns the sequence of intermediary and final results.
singleton : 'T → seq<'T>	Returns a sequence that yields one item only.
skip : int → seq<'T> → seq<'T>	Returns a sequence that skips a specified number of elements of the underlying sequence and then yields the remaining elements of the sequence.
skipWhile : ('T → bool) → seq<'T> → seq<'T>	Returns a sequence that, when iterated, skips elements of the underlying sequence while the given predicate returns **true,** and then yields the remaining elements of the sequence.
sort : seq<'T> → seq<'T>	Yields a sequence ordered by keys.
sortBy : ('T → 'Key) → seq<'T> → seq<'T>	Applies a key-generating function to each element of a sequence and yield a sequence ordered by keys. The keys are compared using generic comparison as implemented by Operators.compare.

sum : seq<^T> → ^T	Returns the sum of the elements in the sequence.
sumBy	Returns the sum of the results generated by applying the function to each element of the sequence.
take : int → seq<'T> → seq<'T>	Returns the first elements of the sequence up to a specified count.
takeWhile : ('T → bool) → seq<'T> → seq<'T>	Returns a sequence that, when iterated, yields elements of the underlying sequence while the given predicate returns **true,** and then returns no further elements.
toArray : seq<'T> → 'T[]	Creates an array from the given collection.
toList : seq<'T> → 'T list	Creates a list from the given collection.
truncate : int → seq<'T> → seq<'T>	Returns a sequence that when enumerated returns no more than a specified number of elements.
tryFind : ('T → bool) → seq<'T> → 'T option	Returns the first element for which the given function returns **true,** or **None** if no such element exists.
tryFindIndex : ('T → bool) → seq<'T> → int option	Returns the index of the first element in the sequence that satisfies the given predicate, or **None** if no such element exists.
tryPick : ('T → 'U option) → seq<'T> → 'U option	Applies the given function to successive elements, returning the first value where the function returns a **Some** value.
unfold : ('State → 'T * 'State option) → 'State → seq<'T>	Returns a sequence that contains the elements generated by the given computation.
where : ('T → bool) → seq<'T> → seq<'T>	Returns a new collection containing only the elements of the collection for which the given predicate returns **true**. A synonym for Seq.filter.
windowed : int → seq<'T> → seq<'T []>	Returns a sequence that yields sliding windows of containing elements drawn from the input sequence. Each window is returned as a fresh array.
zip : seq<'T1> → seq<'T2> → seq<'T1 * 'T2>	Combines the two sequences into a list of pairs. The two sequences need not have equal lengths − when one sequence is exhausted any remaining elements in the other sequence are ignored.
zip3 : seq<'T1> → seq<'T2> → seq<'T3> → seq<'T1 * 'T2 * 'T3>	Combines the three sequences into a list of triples. The sequences need not have equal lengths − when one sequence is exhausted any remaining elements in the other sequences are ignored.

The following examples demonstrate the uses of some of the above functionalities −

Example 1

This program creates an empty sequence and fills it up later −

```
(* Creating sequences *)
let emptySeq = Seq.empty
let seq1 = Seq.singleton 20

printfn"The singleton sequence:"
printfn "%A " seq1
printfn"The init sequence:"

let seq2 = Seq.init 5 (fun n -> n * 3)
Seq.iter (fun i -> printf "%d " i) seq2
printfn""

(* converting an array to sequence by using cast *)
printfn"The array sequence 1:"
let seq3 = [| 1 .. 10 |] :> seq<int>
Seq.iter (fun i -> printf "%d " i) seq3
printfn""

(* converting an array to sequence by using Seq.ofArray *)
printfn"The array sequence 2:"
let seq4 = [| 2..2.. 20 |] |> Seq.ofArray
Seq.iter (fun i -> printf "%d " i) seq4
printfn""
```

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

```
The singleton sequence:
seq [20]
The init sequence:
0 3 6 9 12
The array sequence 1:
1 2 3 4 5 6 7 8 9 10
The array sequence 2:
2 4 6 8 10 12 14 16 18 20
```

Please note that –

- The Seq.empty method creates an empty sequence.
- The Seq.singleton method creates a sequence of just one specified element.
- The Seq.init method creates a sequence for which the elements are created by using a given function.
- The Seq.ofArray and Seq.ofList<'T> methods create sequences from arrays and lists.
- The Seq.iter method allows iterating through a sequence.

Example 2

The Seq.unfold method generates a sequence from a computation function that takes a state and transforms it to produce each subsequent element in the sequence.

The following function produces the first 20 natural numbers –

```
let seq1 = Seq.unfold (fun state -> if (state > 20) then None else Some(state, state + 1))
0
printfn "The sequence seq1 contains numbers from 0 to 20."
for x in seq1 do printf "%d " x
printfn" "
```

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

```
The sequence seq1 contains numbers from 0 to 20.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Example 3

The Seq.truncate method creates a sequence from another sequence, but limits the sequence to a specified number of elements.

The Seq.take method creates a new sequence that contains a specified number of elements from the start of a sequence.

```
let mySeq = seq { for i in 1 .. 10 -> 3*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takeSeq = Seq.take 5 mySeq

printfn"The original sequence"
Seq.iter (fun i -> printf "%d " i) mySeq
printfn""

printfn"The truncated sequence"
Seq.iter (fun i -> printf "%d " i) truncatedSeq
printfn""

printfn"The take sequence"
Seq.iter (fun i -> printf "%d " i) takeSeq
printfn""
```

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

```
The original sequence
3 6 9 12 15 18 21 24 27 30
The truncated sequence
3 6 9 12 15
The take sequence
3 6 9 12 15
```

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 → 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 → 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 → 'T → 'State) → 'State → Set<'T> → 'State	Applies the given accumulating function to all the elements of the set.
foldBack : ('T → 'State → 'State) → Set<'T> → 'State → 'State	Applies the given accumulating function to all the elements of the set.
forall : ('T → 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 → unit) → Set<'T> → unit	Applies the given function to each element of the set, in order according to the comparison function.
map : ('T → '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 → 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
```

F# - MAPS

In F#, a map is a special kind of set that associates the values with key. A map is created in a similar way as sets are created.

Creating Maps

Maps are created by creating an empty map using Map.empty and adding items using the Add function. The following example demonstrates this −

Example

```
(* Create an empty Map *)
let students =
   Map.empty. (* Creating an empty Map *)
      Add("Zara Ali", "1501").
      Add("Rishita Gupta", "1502").
      Add("Robin Sahoo", "1503").
      Add("Gillian Megan", "1504");;
printfn "Map - students: %A" students

(* Convert a list to Map *)
let capitals =
   [ "Argentina", "Buenos Aires";
      "France ", "Paris";
      "Chili", "Santiago";
      "Malaysia", " Kuala Lumpur";
      "Switzerland", "Bern" ]
   |> Map.ofList;;
printfn "Map capitals : %A" capitals
```

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

```
Map - students: map
[("Gillian Megan", "1504"); ("Rishita Gupta", "1502"); ("Robin Sahoo", "1503
");
("Zara Ali", "1501")]
Map capitals : map
[("Argentina", "Buenos Aires"); ("Chili", "Santiago"); ("France ", "Paris");
("Malaysia", " Kuala Lumpur"); ("Switzerland", "Bern")]
```

You can access individual elements in the map using the key.

Example

```
(* Create an empty Map *)
let students =
   Map.empty. (* Creating an empty Map *)
```

```
      Add("Zara Ali", "1501").
      Add("Rishita Gupta", "1502").
      Add("Robin Sahoo", "1503").
      Add("Gillian Megan", "1504");;
printfn "Map - students: %A" students

(*Accessing an element using key *)
printfn "%A" students.["Zara Ali"]
```

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

```
Map - students: map
[("Gillian Megan", "1504"); ("Rishita Gupta", "1502"); ("Robin Sahoo", "1503
");
("Zara Ali", "1501")]
"1501"
```

Basic Operations on Maps

Add module name

The following table shows the basic operations on maps −

Member	Description
Add	Returns a new map with the binding added to the given map.
ContainsKey	Tests if an element is in the domain of the map.
Count	The number of bindings in the map.
IsEmpty	Returns true if there are no bindings in the map.
Item	Lookup an element in the map. Raises KeyNotFoundException if no binding exists in the map.
Remove	Removes an element from the domain of the map. No exception is raised if the element is not present.
TryFind	Lookup an element in the map, returning a **Some** value if the element is in the domain of the map and **None** if not.

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

Example

```
(* Create an empty Map *)
let students =
   Map.empty. (* Creating an empty Map *)
      Add("Zara Ali", "1501").
      Add("Rishita Gupta", "1502").
      Add("Robin Sahoo", "1503").
      Add("Gillian Megan", "1504").
      Add("Shraddha Dubey", "1505").
      Add("Novonil Sarker", "1506").
      Add("Joan Paul", "1507");;
printfn "Map - students: %A" students
printfn "Map - number of students: %d" students.Count

(* finding the registration number of a student*)
let found = students.TryFind "Rishita Gupta"
match found with
| Some x -> printfn "Found %s." x
| None -> printfn "Did not find the specified value."
```

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

```
Map - students: map
[("Gillian Megan", "1504"); ("Joan Paul", "1507"); ("Novonil Sarker", "1506"
);
("Rishita Gupta", "1502"); ("Robin Sahoo", "1503");
("Shraddha Dubey", "1505"); ("Zara Ali", "1501")]
Map - number of students: 7
Found 1502.
```

F# - DISCRIMINATED UNIONS

Unions, or discriminated unions allows you to build up complex data structures representing well-defined set of choices. For example, you need to build an implementation of a *choice* variable, which has two values yes and no. Using the Unions tool, you can design this.

Syntax

Discriminated unions are defined using the following syntax −

```
type type-name =
   | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ]
type2 ...]
   | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4 ...]
...
```

Our simple implementation of ,*choice,* will look like the following −

```
type choice =
   | Yes
   | No
```

The following example uses the type choice −

```
type choice =
   | Yes
   | No

let x = Yes (* creates an instance of choice *)
let y = No (* creates another instance of choice *)
let main() =
   printfn "x: %A" x
   printfn "y: %A" y
main()
```

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

```
x: Yes
y: No
```

Example 1

The following example shows the implementation of the voltage states that sets a bit on high or low −

```
type VoltageState =
   | High
   | Low

let toggleSwitch = function (* pattern matching input *)
   | High -> Low
   | Low -> High
```

```
let main() =
   let on = High
   let off = Low
   let change = toggleSwitch off

   printfn "Switch on state: %A" on
   printfn "Switch off state: %A" off
   printfn "Toggle off: %A" change
   printfn "Toggle the Changed state: %A" (toggleSwitch change)

main()
```

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

```
Switch on state: High
Switch off state: Low
Toggle off: High
Toggle the Changed state: Low
```

Example 2

```
type Shape =
   // here we store the radius of a circle
   | Circle of float

   // here we store the side length.
   | Square of float

   // here we store the height and width.
   | Rectangle of float * float

let pi = 3.141592654

let area myShape =
   match myShape with
   | Circle radius -> pi * radius * radius
   | Square s -> s * s
   | Rectangle (h, w) -> h * w

let radius = 12.0
let myCircle = Circle(radius)
printfn "Area of circle with radius %g: %g" radius (area myCircle)

let side = 15.0
let mySquare = Square(side)
printfn "Area of square that has side %g: %g" side (area mySquare)

let height, width = 5.0, 8.0
let myRectangle = Rectangle(height, width)
printfn "Area of rectangle with height %g and width %g is %g" height width (area
myRectangle)
```

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

```
Area of circle with radius 12: 452.389
Area of square that has side 15: 225
Area of rectangle with height 5 and width 8 is 40
```

F# - MUTABLE DATA

Variables in F# are **immutable,** which means once a variable is bound to a value, it can't be changed. They are actually compiled as static read-only properties.

The following example demonstrates this.

Example

```
let x = 10
let y = 20
let z = x + y

printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z

let x = 15
let y = 20
let z = x + y

printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z
```

When you compile and execute the program, it shows the following error message −

```
Duplicate definition of value 'x'
Duplicate definition of value 'Y'
Duplicate definition of value 'Z'
```

Mutable Variables

At times you need to change the values stored in a variable. To specify that there could be a change in the value of a declared and assigned variable in later part of a program, F# provides the **mutable** keyword. You can declare and assign mutable variables using this keyword, whose values you will change.

The **mutable** keyword allows you to declare and assign values in a mutable variable.

You can assign some initial value to a mutable variable using the **let** keyword. However, to assign new subsequent value to it, you need to use the **<-** operator.

For example,

```
let mutable x = 10
x <- 15
```

The following example will clear the concept −

Example

```
let mutable x = 10
let y = 20
let mutable z = x + y

printfn "Original Values:"
printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z

printfn "Let us change the value of x"
printfn "Value of z will change too."

x <- 15
z <- x + y

printfn "New Values:"
printfn "x: %i" x
printfn "y: %i" y
printfn "z: %i" z
```

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

```
Original Values:
x: 10
y: 20
z: 30
Let us change the value of x
Value of z will change too.
New Values:
x: 15
y: 20
z: 35
```

Uses of Mutable Data

Mutable data is often required and used in data processing, particularly with record data structure. The following example demonstrates this −

```
open System

type studentData =
   { ID : int;
      mutable IsRegistered : bool;
      mutable RegisteredText : string; }

let getStudent id =
   { ID = id;
      IsRegistered = false;
      RegisteredText = null; }

let registerStudents (students : studentData list) =
   students |> List.iter(fun st ->
      st.IsRegistered <- true
      st.RegisteredText <- sprintf "Registered %s" (DateTime.Now.ToString("hh:mm:ss"))

      Threading.Thread.Sleep(1000) (* Putting thread to sleep for 1 second to simulate
processing overhead. *))

let printData (students : studentData list) =
   students |> List.iter (fun x -> printfn "%A" x)

let main() =
   let students = List.init 3 getStudent

   printfn "Before Process:"
   printData students

   printfn "After process:"
   registerStudents students
   printData students

   Console.ReadKey(true) |> ignore

main()
```

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

```
Before Process:
{ID = 0;
IsRegistered = false;
RegisteredText = null;}
{ID = 1;
IsRegistered = false;
RegisteredText = null;}
{ID = 2;
IsRegistered = false;
RegisteredText = null;}
After process:
{ID = 0;
IsRegistered = true;
```

```
RegisteredText = "Registered 05:39:15";}
{ID = 1;
IsRegistered = true;
RegisteredText = "Registered 05:39:16";}
{ID = 2;
IsRegistered = true;
RegisteredText = "Registered 05:39:17";}
```

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 → ^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 → U option) → '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 → '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 → bool) → 'T [] → bool	Tests whether any element of an array satisfies the supplied predicate.
exists2 : ('T1 → 'T2 → 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 → bool) → 'T [] → 'T []	Returns a collection that contains only the elements of the supplied array for which the supplied condition returns **true**.
find : ('T → bool) → 'T [] → 'T	Returns the first element for which the supplied function returns **true**. Raises KeyNotFoundException if no such element exists.
findIndex : ('T → 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 → 'T → '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 (...(f s i0)...) iN.

fold2 : ('State → 'T1 → 'T2 → '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 → 'State → '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 (...(f iN s)).
foldBack2 : ('T1 → 'T2 → 'State → '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 → bool) → 'T [] → bool	Tests whether all elements of an array satisfy the supplied condition.
forall2 : ('T1 → 'T2 → 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 → '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 → unit) → 'T [] → unit	Applies the supplied function to each element of an array.
iter2 : ('T1 → 'T2 → 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 → 'T → 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 → 'T1 → 'T2 → 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 → '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 → 'T2 → '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 → 'T → '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 → 'T1 → 'T2 → '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 → '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 → '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 → 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 → int) → 'T [] → 'T []	Permutes the elements of an array according to the specified permutation.
pick : ('T → 'U option) → '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 → 'T → '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 (...(f i0 i1)...) iN. If the array has size zero, ArgumentException is raised.
reduceBack : ('T → 'T → '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 (...(f iN-1 iN)). If the array has size zero, ArgumentException is raised.
rev : 'T [] → 'T []	Reverses the order of the elements in a supplied array.
scan : ('State → 'T → 'State) → 'State → 'T [] → 'State [])	Behaves like fold, but returns the intermediate results together with the final results.
scanBack : ('T → 'State → '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 → '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 → '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 → 'T → int) → 'T [] → unit	Sorts the elements of an array by using the supplied comparison function to change the array in place.
sortWith : ('T → 'T → 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 → ^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 → 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 → bool) → 'T [] → int option	Returns the index of the first element in an array that satisfies the supplied condition.
tryPick : ('T → 'U option) → '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.
```

F# - MUTABLE LISTS

The **List<'T>** class represents a strongly typed list of objects that can be accessed by index.

It is a mutable counterpart of the List class. It is similar to arrays, as it can be accessed by an index, however, unlike arrays, lists can be resized. Therefore you need not specify a size during declaration.

Creating a Mutable List

Lists are created using the **new** keyword and calling the list's constructor. The following example demonstrates this −

```
(* Creating a List *)
open System.Collections.Generic

let booksList = new List<string>()
booksList.Add("Gone with the Wind")
booksList.Add("Atlas Shrugged")
booksList.Add("Fountainhead")
booksList.Add("Thornbirds")
booksList.Add("Rebecca")
booksList.Add("Narnia")

booksList |> Seq.iteri (fun index item -> printfn "%i: %s" index booksList.[index])
```

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

```
0: Gone with the Wind
1: Atlas Shrugged
2: Fountainhead
3: Thornbirds
4: Rebecca
5: Narnia
```

The List(T) Class

The List(T) class represents a strongly typed list of objects that can be accessed by index. It provide methods to search, sort, and manipulate lists.

The following tables provide the properties, constructors and the methods of the List(T) class −

Properties

Property	Description
Capacity	Gets or sets the total number of elements the internal data structure can hold without resizing.
Count	Gets the number of elements contained in the List(T).
Item	Gets or sets the element at the specified index.

Constructors

Constructor	Description
List(T)()	Initializes a new instance of the List(T) class that is empty and has the default initial capacity.
List(T)(IEnumerable(T))	Initializes a new instance of the List(T) class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.
List(T)(Int32)	Initializes a new instance of the List(T) class that is empty and has the specified initial capacity.

Method

Methods	Description
Add	Adds an object to the end of the List(T).
AddRange	Adds the elements of the specified collection to the end of the List(T).
AsReadOnly	Returns a read-only IList(T) wrapper for the current collection.
BinarySearch(T)	Searches the entire sorted List(T) for an element using the default comparer and returns the zero-based index of the element.
BinarySearch(T, IComparer(T))	Searches the entire sorted List(T) for an element using the specified comparer and returns the zero-based index of the element.
BinarySearch(Int32, Int32, T, IComparer(T))	Searches a range of elements in the sorted List(T) for an element using the specified comparer and returns the zero-based index of the element.
Clear	Removes all elements from the List(T).
Contains	Determines whether an element is in the List(T).
ConvertAll(TOutput)	Converts the elements in the current List(T) to another type, and returns a list containing the converted elements.
CopyTo(T[])	Copies the entire List(T) to a compatible one-

	dimensional array, starting at the beginning of the target array.
CopyTo(T[], Int32)	Copies the entire List(T) to a compatible one-dimensional array, starting at the specified index of the target array.
CopyTo(Int32, T[], Int32, Int32)	Copies a range of elements from the List(T) to a compatible one-dimensional array, starting at the specified index of the target array.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object.)
Exists	Determines whether the List(T) contains elements that match the conditions defined by the specified predicate.
Finalize	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection (Inherited from Object).
Find	Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List(T).
FindAll	Retrieves all the elements that match the conditions defined by the specified predicate.
FindIndex(Predicate(T))	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the entire List(T).
FindIndex(Int32, Predicate(T))	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the range of elements in the List(T) that extends from the specified index to the last element.
FindIndex(Int32, Int32, Predicate(T))	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the range of elements in the List(T) that starts at the specified index and contains the specified number of elements.
FindLast	Searches for an element that matches the conditions defined by the specified predicate, and returns the last occurrence within the entire List(T).
FindLastIndex(Predicate(T))	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the entire List(T).
FindLastIndex(Int32, Predicate(T))	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the range of elements in the List(T) that extends from the first element to the specified index.
FindLastIndex(Int32, Int32, Predicate(T))	Searches for an element that matches the conditions defined by the specified predicate, and

	returns the zero-based index of the last occurrence within the range of elements in the List(T) that contains the specified number of elements and ends at the specified index.
ForEach	Performs the specified action on each element of the List(T).
GetEnumerator	Returns an enumerator that iterates through the List(T).
GetHashCode	Serves as the default hash function. (Inherited from Object.)
GetRange	Creates a shallow copy of a range of elements in the source List(T).
GetType	Gets the Type of the current instance. (Inherited from Object.)
IndexOf(T)	Searches for the specified object and returns the zero-based index of the first occurrence within the entire List(T).
IndexOf(T, Int32)	Searches for the specified object and returns the zero-based index of the first occurrence within the range of elements in the List(T) that extends from the specified index to the last element.
IndexOf(T, Int32, Int32)	Searches for the specified object and returns the zero-based index of the first occurrence within the range of elements in the List(T) that starts at the specified index and contains the specified number of elements.
Insert	Inserts an element into the List(T) at the specified index.
InsertRange	Inserts the elements of a collection into the List(T) at the specified index.
LastIndexOf(T)	Searches for the specified object and returns the zero-based index of the last occurrence within the entire List(T).
LastIndexOf(T, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List(T) that extends from the first element to the specified index.
LastIndexOf(T, Int32, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List(T) that contains the specified number of elements and ends at the specified index.
MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
Remove	Removes the first occurrence of a specific object from the List(T).
RemoveAll	Removes all the elements that match the conditions defined by the specified predicate.
RemoveAt	Removes the element at the specified index of the List(T).

RemoveRange	Removes a range of elements from the List(T).
Reverse()	Reverses the order of the elements in the entire List(T).
Reverse(Int32, Int32)	Reverses the order of the elements in the specified range.
Sort()	Sorts the elements in the entire List(T) using the default comparer.
Sort(Comparison(T))	Sorts the elements in the entire List(T) using the specified System. Comparison(T).
Sort(IComparer(T))	Sorts the elements in the entire List(T) using the specified comparer.
Sort(Int32, Int32, IComparer(T))	Sorts the elements in a range of elements in List(T) using the specified comparer.
ToArray	Copies the elements of the List(T) to a new array.
ToString	Returns a string that represents the current object. (Inherited from Object.)
TrimExcess	Sets the capacity to the actual number of elements in the List(T), if that number is less than a threshold value.
TrueForAll	Determines whether every element in the List(T) matches the conditions defined by the specified predicate.

Example

```
(* Creating a List *)
open System.Collections.Generic

let booksList = new List<string>()
booksList.Add("Gone with the Wind")
booksList.Add("Atlas Shrugged")
booksList.Add("Fountainhead")
booksList.Add("Thornbirds")
booksList.Add("Rebecca")
booksList.Add("Narnia")

printfn"Total %d books" booksList.Count
booksList |> Seq.iteri (fun index item -> printfn "%i: %s" index booksList.[index])
booksList.Insert(2, "Roots")

printfn("after inserting at index 2")
printfn"Total %d books" booksList.Count

booksList |> Seq.iteri (fun index item -> printfn "%i: %s" index booksList.[index])
booksList.RemoveAt(3)

printfn("after removing from index 3")
printfn"Total %d books" booksList.Count

booksList |> Seq.iteri (fun index item -> printfn "%i: %s" index booksList.[index])
```

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

```
Total 6 books
0: Gone with the Wind
```

```
1: Atlas Shrugged
2: Fountainhead
3: Thornbirds
4: Rebecca
5: Narnia
after inserting at index 2
Total 7 books
0: Gone with the Wind
1: Atlas Shrugged
2: Roots
3: Fountainhead
4: Thornbirds
5: Rebecca
6: Narnia
after removing from index 3
Total 6 books
0: Gone with the Wind
1: Atlas Shrugged
2: Roots
3: Thornbirds
4: Rebecca
5: Narnia
```

F# - MUTABLE DICTIONARY

The **Dictionary<'TKey, 'TValue>** class is the mutable analog of the F# map data structure and contains many of the same functions.

Recapitulating from the Map chapter in F#, a map is a special kind of set that associates the values with key.

Creating of a Mutable Dictionary

Mutable dictionaries are created using the **new** keyword and calling the list's constructor. The following example demonstrates this −

```
open System.Collections.Generic
let dict = new Dictionary<string, string>()
dict.Add("1501", "Zara Ali")
dict.Add("1502","Rishita Gupta")
dict.Add("1503","Robin Sahoo")
dict.Add("1504","Gillian Megan")
printfn "Dictionary - students: %A" dict
```

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

```
Dictionary - students: seq
[[1501, Zara Ali]; [1502, Rishita Gupta]; [1503, Robin Sahoo];
[1504, Gillian Megan]]
```

The Dictionary(TKey,TValue) Class

The Dictionary(TKey, TValue) Class represents a collection of keys and values.

The following tables provide the properties, constructors and the methods of the List(T) class −

Properties

Property	Description
Comparer	Gets the IEqualityComparer(T) that is used to determine equality of keys for the dictionary.
Count	Gets the number of key/value pairs contained in the Dictionary(TKey, TValue).

Item	Gets or sets the value associated with the specified key.
Keys	Gets a collection containing the keys in the Dictionary(TKey, TValue).
Values	Gets a collection containing the values in the Dictionary(TKey, TValue).

Constructors

Constructors	Description
Dictionary(TKey, TValue)()	Initializes a new instance of the **Dictionary(TKey, TValue)** class that is empty, has the default initial capacity, and uses the default equality comparer for the key type.
Dictionary(TKey, TValue)(IDictionary(TKey, TValue))	Initializes a new instance of the **Dictionary(TKey, TValue)** class that contains elements copied from the specified **IDictionary(TKey, TValue)** and uses the default equality comparer for the key type.
Dictionary(TKey, TValue)(IEqualityComparer(TKey))	Initializes a new instance of the **Dictionary(TKey, TValue)** class that is empty, has the default initial capacity, and uses the specified **IEqualityComparer(T).**
Dictionary(TKey, TValue)(Int32)	Initializes a new instance of the **Dictionary(TKey, TValue)** class that is empty, has the specified initial capacity, and uses the default equality comparer for the key type.
Dictionary(TKey, TValue)(IDictionary(TKey, TValue), IEqualityComparer(TKey))	Initializes a new instance of the **Dictionary(TKey, TValue)** class that contains elements copied from the specified **IDictionary(TKey, TValue)** and uses the specified **IEqualityComparer(T).**
Dictionary(TKey, TValue)(Int32, IEqualityComparer(TKey))	Initializes a new instance of the **Dictionary(TKey, TValue)** class that is empty, has the specified initial capacity, and uses the specified **IEqualityComparer(T).**
Dictionary(TKey, TValue)(SerializationInfo, StreamingContext)	Initializes a new instance of the **ictionary(TKey, TValue)** class with serialized data.

Methods

Method	Description
Add	Adds the specified key and value to the dictionary.
Clear	Removes all keys and values from the Dictionary(TKey, TValue).
ContainsKey	Determines whether the Dictionary(TKey, TValue) contains the specified key.
ContainsValue	Determines whether the Dictionary(TKey, TValue) contains a specific value.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object.)
Finalize	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)
GetEnumerator	Returns an enumerator that iterates through the Dictionary(TKey, TValue).

GetHashCode	Serves as the default hash function. (Inherited from Object.)
GetObjectData	Implements the System.Runtime.Serialization.ISerializable interface and returns the data needed to serialize the Dictionary(TKey, TValue)instance.
GetType	Gets the Type of the current instance. (Inherited from Object.)
MemberwiseClone	Creates a shallow copy of the current Object. (Inherited from Object.)
OnDeserialization	Implements the System.Runtime.Serialization.ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove	Removes the value with the specified key from the Dictionary(TKey, TValue).
ToString	Returns a string that represents the current object. (Inherited from Object.)
TryGetValue	Gets the value associated with the specified key.

Example

```
open System.Collections.Generic
let dict = new Dictionary<string, string>()

dict.Add("1501", "Zara Ali")
dict.Add("1502","Rishita Gupta")
dict.Add("1503","Robin Sahoo")
dict.Add("1504","Gillian Megan")

printfn "Dictionary - students: %A" dict
printfn "Total Number of Students: %d" dict.Count
printfn "The keys: %A" dict.Keys
printf"The Values: %A" dict.Values
```

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

```
Dictionary - students: seq
[[1501, Zara Ali]; [1502, Rishita Gupta]; [1503, Robin Sahoo];
[1504, Gillian Megan]]
Total Number of Students: 4
The keys: seq ["1501"; "1502"; "1503"; "1504"]
The Values: seq ["Zara Ali"; "Rishita Gupta"; "Robin Sahoo"; "Gillian Megan"]
```

F# - BASIC IO

Basic Input Output includes −

- Reading from and writing into console.
- Reading from and writing into file.

Core.Printf Module

We have used the *printf* and the *printfn* functions for writing into the console. In this section, we will look into the details of the **Printf** module of F#.

Apart from the above functions, the *Core.Printf* module of F# has various other methods for printing and formatting using % markers as placeholders. The following table shows the methods with brief description −

Value	Description
bprintf : StringBuilder → BuilderFormat<'T> → 'T	Prints to a StringBuilder.

eprintf : TextWriterFormat<'T> → 'T	Prints formatted output to stderr.
eprintfn : TextWriterFormat<'T> → 'T	Prints formatted output to stderr, adding a newline.
failwithf : StringFormat<'T,'Result> → 'T	Prints to a string buffer and raises an exception with the given result.
fprintf : TextWriter → TextWriterFormat<'T> → 'T	Prints to a text writer.
fprintfn : TextWriter → TextWriterFormat<'T> → 'T	Prints to a text writer, adding a newline.
kbprintf : (unit → 'Result) → StringBuilder → BuilderFormat<'T,'Result> → 'T	Like bprintf, but calls the specified function to generate the result.
kfprintf : (unit → 'Result) → TextWriter → TextWriterFormat<'T,'Result> → 'T	Like fprintf, but calls the specified function to generate the result.
kprintf : (string → 'Result) → StringFormat<'T,'Result> → 'T	Like printf, but calls the specified function to generate the result. For example, these let the printing force a flush after all output has been entered onto the channel, but not before.
ksprintf : (string → 'Result) → StringFormat<'T,'Result> → 'T	Like sprintf, but calls the specified function to generate the result.
printf : TextWriterFormat<'T> → 'T	Prints formatted output to stdout.
printfn : TextWriterFormat<'T> → 'T	Prints formatted output to stdout, adding a newline.
sprintf : StringFormat<'T> → 'T	Prints to a string by using an internal string buffer and returns the result as a string.

Format Specifications

Format specifications are used for formatting the input or output, according to the programmers' need.

These are strings with % markers indicating format placeholders.

The syntax of a Format placeholders is −

```
%[flags][width][.precision][type]
```

The **type** is interpreted as −

Type	Description
%b	Formats a **bool,** formatted as **true** or **false**.
%c	Formats a character.
%s	Formats a **string,** formatted as its contents, without interpreting any escape characters.
%d, %i	Formats any basic integer type formatted as a decimal integer, signed if the basic integer type is signed.
%u	Formats any basic integer type formatted as an unsigned decimal integer.

%x	Formats any basic integer type formatted as an unsigned hexadecimal integer, using lowercase letters a through f.
%X	Formats any basic integer type formatted as an unsigned hexadecimal integer, using uppercase letters A through F.
%o	Formats any basic integer type formatted as an unsigned octal integer.
%e, %E, %f, %F, %g, %G	Formats any basic floating point type **(float, float32)** formatted using a C-style floating point format specifications.
%e, %E	Formats a signed value having the form [-]d.ddde[sign]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or -.
%f	Formats a signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
%g, %G	Formats a signed value printed in f or e format, whichever is more compact for the given value and precision.
%M	Formats a Decimal value.
%O	Formats any value, printed by boxing the object and using its **ToString** method.
%A, %+A	Formats any value, printed with the default layout settings. Use %+A to print the structure of discriminated unions with internal and private representations.
%a	A general format specifier, requires two arguments. The first argument is a function which accepts two arguments: first, a context parameter of the appropriate type for the given formatting function (for example, a TextWriter), and second, a value to print and which either outputs or returns appropriate text. The second argument is the particular value to print.
%t	A general format specifier, requires one argument: a function which accepts a context parameter of the appropriate type for the given formatting function (aTextWriter) and which either outputs or returns appropriate text. Basic integer types are **byte, sbyte, int16, uint16, int32, uint32, int64, uint64, nativeint,** and **unativeint.** Basic floating point types are **float** and **float32.**

The **width** is an optional parameter. It is an integer that indicates the minimal width of the result. For example, %5d prints an integer with at least spaces of 5 characters.

Valid **flags** are described in the following table −

Value	Description
0	Specifies to add zeros instead of spaces to make up the required width.
-	Specifies to left-justify the result within the width specified.
+	Specifies to add a + character if the number is positive (to match a - sign for negative numbers).
' '	Specifies to add an extra space if the number is positive (to match a - sign for

(space)	negative numbers).
#	Invalid.

Example

```
printf "Hello "
printf "World"
printfn ""
printfn "Hello "
printfn "World"
printf "Hi, I'm %s and I'm a %s" "Rohit" "Medical Student"

printfn "d: %f" 212.098f
printfn "e: %f" 504.768f

printfn "x: %g" 212.098f
printfn "y: %g" 504.768f

printfn "x: %e" 212.098f
printfn "y: %e" 504.768f
printfn "True: %b" true
```

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

```
Hello World
Hello
World
Hi, I'm Rohit and I'm a Medical Studentd: 212.098000
e: 504.768000
x: 212.098
y: 504.768
x: 2.120980e+002
y: 5.047680e+002
True: true
```

The Console Class

This class is a part of the .NET framework. It represents the standard input, output, and error streams for console applications.

It provides various methods for reading from and writing into the console. The following table shows the methods −

Method	Description
Beep()	Plays the sound of a beep through the console speaker.
Beep(Int32, Int32)	Plays the sound of a beep of a specified frequency and duration through the console speaker.
Clear	Clears the console buffer and corresponding console window of display information.
MoveBufferArea(Int32, Int32, Int32, Int32, Int32, Int32)	Copies a specified source area of the screen buffer to a specified destination area.
MoveBufferArea(Int32, Int32, Int32, Int32, Int32, Int32, Char, ConsoleColor, ConsoleColor)	Copies a specified source area of the screen buffer to a specified destination area.
OpenStandardError()	Acquires the standard error stream.

OpenStandardError(Int32)	Acquires the standard error stream, which is set to a specified buffer size.
OpenStandardInput()	Acquires the standard input stream.
OpenStandardInput(Int32)	Acquires the standard input stream, which is set to a specified buffer size.
OpenStandardOutput()	Acquires the standard output stream.
OpenStandardOutput(Int32)	Acquires the standard output stream, which is set to a specified buffer size.
Read	Reads the next character from the standard input stream.
ReadKey()	Obtains the next character or function key pressed by the user. The pressed key is displayed in the console window.
ReadKey(Boolean)	Obtains the next character or function key pressed by the user. The pressed key is optionally displayed in the console window.
ReadLine	Reads the next line of characters from the standard input stream.
ResetColor	Sets the foreground and background console colors to their defaults.
SetBufferSize	Sets the height and width of the screen buffer area to the specified values.
SetCursorPosition	Sets the position of the cursor.
SetError	Sets the Error property to the specified TextWriter object.
SetIn	Sets the In property to the specified TextReader object.
SetOut	Sets the Out property to the specified TextWriter object.
SetWindowPosition	Sets the position of the console window relative to the screen buffer.
SetWindowSize	Sets the height and width of the console window to the specified values.
Write(Boolean)	Writes the text representation of the specified Boolean value to the standard output stream.
Write(Char)	Writes the specified Unicode character value to the standard output stream.
Write(Char[])	Writes the specified array of Unicode characters to the standard output stream.
Write(Decimal)	Writes the text representation of the specified Decimal value to the standard output stream.
Write(Double)	Writes the text representation of the specified double-precision floating-point value to the standard output stream.
Write(Int32)	Writes the text representation of the specified 32-bit signed integer value to the standard output stream.
Write(Int64)	Writes the text representation of the specified 64-bit signed integer value to the standard output stream.
Write(Object)	Writes the text representation of the specified object to the standard output stream.

Write(Single)	Writes the text representation of the specified single-precision floating-point value to the standard output stream.
Write(String)	Writes the specified string value to the standard output stream.
Write(UInt32)	Writes the text representation of the specified 32-bit unsigned integer value to the standard output stream.
Write(UInt64)	Writes the text representation of the specified 64-bit unsigned integer value to the standard output stream.
Write(String, Object)	Writes the text representation of the specified object to the standard output stream using the specified format information.
Write(String, Object[])	Writes the text representation of the specified array of objects to the standard output stream using the specified format information.
Write(Char[], Int32, Int32)	Writes the specified subarray of Unicode characters to the standard output stream.
Write(String, Object, Object)	Writes the text representation of the specified objects to the standard output stream using the specified format information.
Write(String, Object, Object, Object)	Writes the text representation of the specified objects to the standard output stream using the specified format information.
Write(String, Object, Object, Object, Object)	Writes the text representation of the specified objects and variable-length parameter list to the standard output stream using the specified format information.
WriteLine()	Writes the current line terminator to the standard output stream.
WriteLine(Boolean)	Writes the text representation of the specified Boolean value, followed by the current line terminator, to the standard output stream.
WriteLine(Char)	Writes the specified Unicode character, followed by the current line terminator, value to the standard output stream.
WriteLine(Char[])	Writes the specified array of Unicode characters, followed by the current line terminator, to the standard output stream.
WriteLine(Decimal)	Writes the text representation of the specified Decimal value, followed by the current line terminator, to the standard output stream.
WriteLine(Double)	Writes the text representation of the specified double-precision floating-point value, followed by the current line terminator, to the standard output stream.
WriteLine(Int32)	Writes the text representation of the specified 32-bit signed integer value, followed by the current line terminator, to the standard output stream.
WriteLine(Int64)	Writes the text representation of the specified 64-bit signed integer value, followed by the current line terminator, to the standard output stream.
WriteLine(Object)	Writes the text representation of the specified object, followed by the current line terminator, to the standard output stream.
WriteLine(Single)	Writes the text representation of the specified single-precision floating-point value, followed by the current line terminator, to the standard output stream.
WriteLine(String)	Writes the specified string value, followed by the current line

	terminator, to the standard output stream.
WriteLine(UInt32)	Writes the text representation of the specified 32-bit unsigned integer value, followed by the current line terminator, to the standard output stream.
WriteLine(UInt64)	Writes the text representation of the specified 64-bit unsigned integer value, followed by the current line terminator, to the standard output stream.
WriteLine(String, Object)	Writes the text representation of the specified object, followed by the current line terminator, to the standard output stream using the specified format information.
WriteLine(String, Object[])	Writes the text representation of the specified array of objects, followed by the current line terminator, to the standard output stream using the specified format information.
WriteLine(Char[], Int32, Int32)	Writes the specified subarray of Unicode characters, followed by the current line terminator, to the standard output stream.
WriteLine(String, Object, Object)	Writes the text representation of the specified objects, followed by the current line terminator, to the standard output stream using the specified format information.
WriteLine(String, Object, Object, Object)	Writes the text representation of the specified objects, followed by the current line terminator, to the standard output stream using the specified format information.
WriteLine(String, Object, Object, Object, Object)	Writes the text representation of the specified objects and variable-length parameter list, followed by the current line terminator, to the standard output stream using the specified format information.

The following example demonstrates reading from console and writing into it −

Example

```
open System
let main() =
   Console.Write("What's your name? ")
   let name = Console.ReadLine()
   Console.Write("Hello, {0}\n", name)
   Console.WriteLine(System.String.Format("Big Greetings from {0} and {1}",
"TutorialsPoint", "Absoulte Classes"))
   Console.WriteLine(System.String.Format("|{0:yyyy-MMM-dd}|", System.DateTime.Now))
main()
```

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

```
What's your name? Kabir
Hello, Kabir
Big Greetings from TutorialsPoint and Absoulte Classes
|2015-Jan-05|
```

The System.IO Namespace

The System.IO namespace contains a variety of useful classes for performing basic I/O.

It contains types or classes that allow reading and writing to files and data streams and types that provide basic file and directory support.

Classes useful for working with the file system −

- The System.IO.File class is used for creating, appending, and deleting files.

- System.IO.Directory class is used for creating, moving, and deleting directories.
- System.IO.Path class performs operations on strings, which represent file paths.
- System.IO.FileSystemWatcher class allows users to listen to a directory for changes.

Classes useful for working with the streams (sequence of bytes) −

- System.IO.StreamReader class is used to read characters from a stream.
- System.IO.StreamWriter class is used to write characters to a stream.
- System.IO.MemoryStream class creates an in-memory stream of bytes.

The following table shows all the classes provided in the namespace along with a brief description −

Class	Description
BinaryReader	Reads primitive data types as binary values in a specific encoding.
BinaryWriter	Writes primitive types in binary to a stream and supports writing strings in a specific encoding.
BufferedStream	Adds a buffering layer to read and write operations on another stream.
Directory	Exposes static methods for creating, moving, and enumerating through directories and subdirectories.
DirectoryInfo	Exposes instance methods for creating, moving, and enumerating through directories and subdirectories.
DirectoryNotFoundException	The exception that is thrown when part of a file or directory cannot be found.
DriveInfo	Provides access to information on a drive.
DriveNotFoundException	The exception that is thrown when trying to access a drive or share that is not available.
EndOfStreamException	The exception that is thrown when reading is attempted past the end of a stream.
ErrorEventArgs	Provides data for the FileSystemWatcher.Error event.
File	Provides static methods for the creation, copying, deletion, moving, and opening of a single file, and aids in the creation of FileStream objects.
FileFormatException	The exception that is thrown when an input file or a data stream that is supposed to conform to a certain file format specification is malformed.
FileInfo	Provides properties and instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects.
FileLoadException	The exception that is thrown when a managed assembly is found but cannot be loaded.
FileNotFoundException	The exception that is thrown when an attempt to access a file that does not exist on disk fails.
FileStream	Exposes a Stream around a file, supporting both synchronous and asynchronous read and write

	operations.
FileSystemEventArgs	Provides data for the directory events − Changed, Created, Deleted.
FileSystemInfo	Provides the base class for both FileInfo and DirectoryInfo objects.
FileSystemWatcher	Listens to the file system change notifications and raises events when a directory, or file in a directory, changes.
InternalBufferOverflowException	The exception thrown when the internal buffer overflows.
InvalidDataException	The exception that is thrown when a data stream is in an invalid format.
IODescriptionAttribute	Sets the description visual designers can display when referencing an event, extender, or property.
IOException	The exception that is thrown when an I/O error occurs.
MemoryStream	Creates a stream whose backing store is memory.
Path	Performs operations on String instances that contain file or directory path information. These operations are performed in a cross-platform manner.
PathTooLongException	The exception that is thrown when a path or file name is longer than the system-defined maximum length.
PipeException	Thrown when an error occurs within a named pipe.
RenamedEventArgs	Provides data for the Renamed event.
Stream	Provides a generic view of a sequence of bytes. This is an abstract class.
StreamReader	Implements a TextReader that reads characters from a byte stream in a particular encoding.
StreamWriter	Implements a TextWriter for writing characters to a stream in a particular encoding. To browse the .NET Framework source code for this type, see the Reference Source.
StringReader	Implements a TextReader that reads from a string.
StringWriter	Implements a TextWriter for writing information to a string. The information is stored in an underlying StringBuilder.
TextReader	Represents a reader that can read a sequential series of characters.
TextWriter	Represents a writer that can write a sequential series of characters. This class is abstract.
UnmanagedMemoryAccessor	Provides random access to unmanaged blocks of memory from managed code.
UnmanagedMemoryStream	Provides access to unmanaged blocks of memory from managed code.
WindowsRuntimeStorageExtensions	Contains extension methods for the IStorageFile and IStorageFolder interfaces in the Windows Runtime when developing Windows Store apps.
WindowsRuntimeStreamExtensions	Contains extension methods for converting between

streams in the Windows Runtime and managed streams in the .NET for Windows Store apps.

Example

The following example creates a file called test.txt, writes a message there, reads the text from the file and prints it on the console.

Note – The amount of code needed to do this is surprisingly less!

```
open System.IO // Name spaces can be opened just as modules
File.WriteAllText("test.txt", "Hello There\n Welcome to:\n Tutorials Point")
let msg = File.ReadAllText("test.txt")
printfn "%s" msg
```

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

```
Hello There
Welcome to:
Tutorials Point
```

F# - GENERICS

Generics allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type.

In F#, function values, methods, properties, and aggregate types such as classes, records, and discriminated unions can be generic.

Generic constructs contain at least one type parameter. Generic functions and types enable you to write code that works with a variety of types without repeating the code for each type.

Syntax

Syntax for writing a generic construct is as follows –

```
// Explicitly generic function.
let function-name<type-parameters> parameter-list =
   function-body

// Explicitly generic method.
[ static ] member object-identifer.method-name<type-parameters> parameter-list [ return-
type ] =
   method-body

// Explicitly generic class, record, interface, structure,
// or discriminated union.
type type-name<type-parameters> type-definition
```

Examples

```
(* Generic Function *)
let printFunc<'T> x y =
   printfn "%A, %A" x y

printFunc<float> 10.0 20.0
```

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

```
10.0, 20.0
```

You can also make a function generic by using the single quotation mark syntax −

```
(* Generic Function *)
let printFunction (x: 'a) (y: 'a) =
   printfn "%A %A" x y

printFunction 10.0 20.0
```

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

```
10.0 20.0
```

Please note that when you use generic functions or methods, you might not have to specify the type arguments. However, in case of an ambiguity, you can provide type arguments in angle brackets as we did in the first example.

If you have more than one type, then you separate multiple type arguments with commas.

Generic Class

Like generic functions, you can also write generic classes. The following example demonstrates this −

```
type genericClass<'a> (x: 'a) =
   do printfn "%A" x

let gr = new genericClass<string>("zara")
let gs = genericClass( seq { for i in 1 .. 10 -> (i, i*i) } )
```

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

```
"zara"
seq [(1, 1); (2, 4); (3, 9); (4, 16); ...]
```

F# - DELEGATES

A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime. F# delegates are similar to pointers to functions, in C or C++.

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which have the same signature as that of the delegate.

Syntax for delegate declaration is −

```
type delegate-typename = delegate of type1 -> type2
```

For example, consider the delegates −

```
// Delegate1 works with tuple arguments.
type Delegate1 = delegate of (int * int) -> int
// Delegate2 works with curried arguments.
type Delegate2 = delegate of int * int -> int
```

Both the delegates can be used to reference any method that has two *int* parameters and returns an *int* type variable.

In the syntax −

- ***type1*** represents the argument type(s).

- ***type2*** represents the return type.

Please note −

- The argument types are automatically curried.
- Delegates can be attached to function values, and static or instance methods.
- F# function values can be passed directly as arguments to delegate constructors.
- For a static method the delegate is called by using the name of the class and the method. For an instance method, the name of the object instance and method is used.
- The Invoke method on the delegate type calls the encapsulated function.
- Also, delegates can be passed as function values by referencing the Invoke method name without the parentheses.

The following example demonstrates the concept −

Example

```
type Myclass() =
   static member add(a : int, b : int) =
      a + b
   static member sub (a : int) (b : int) =
      a - b
   member x.Add(a : int, b : int) =
      a + b
   member x.Sub(a : int) (b : int) =
      a - b

// Delegate1 works with tuple arguments.
type Delegate1 = delegate of (int * int) -> int
// Delegate2 works with curried arguments.
type Delegate2 = delegate of int * int -> int

let InvokeDelegate1 (dlg : Delegate1) (a : int) (b: int) =
   dlg.Invoke(a, b)
let InvokeDelegate2 (dlg : Delegate2) (a : int) (b: int) =
   dlg.Invoke(a, b)

// For static methods, use the class name, the dot operator, and the
// name of the static method.
let del1 : Delegate1 = new Delegate1( Myclass.add )
let del2 : Delegate2 = new Delegate2( Myclass.sub )

let mc = Myclass()
// For instance methods, use the instance value name, the dot operator, and the instance
method name.

let del3 : Delegate1 = new Delegate1( mc.Add )
let del4 : Delegate2 = new Delegate2( mc.Sub )

for (a, b) in [ (400, 200); (100, 45) ] do
   printfn "%d + %d = %d" a b (InvokeDelegate1 del1 a b)
   printfn "%d - %d = %d" a b (InvokeDelegate2 del2 a b)
   printfn "%d + %d = %d" a b (InvokeDelegate1 del3 a b)
   printfn "%d - %d = %d" a b (InvokeDelegate2 del4 a b)
```

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

```
400 + 200 = 600
400 - 200 = 200
400 + 200 = 600
400 - 200 = 200
100 + 45 = 145
```

```
100 - 45 = 55
100 + 45 = 145
100 - 45 = 55
```

F# - ENUMERATIONS

An enumeration is a set of named integer constants.

In F#, **enumerations,** also known as **enums,** are integral types where labels are assigned to a subset of the values. You can use them in place of literals to make code more readable and maintainable.

Declaring Enumerations

The general syntax for declaring an enumeration is −

```
type enum-name =
   | value1 = integer-literal1
   | value2 = integer-literal2
...
```

The following example demonstrates the use of enumerations −

Example

```
// Declaration of an enumeration.
type Days =
   | Sun = 0
   | Mon = 1
   | Tues = 2
   | Wed = 3
   | Thurs = 4
   | Fri = 5
   | Sat = 6

// Use of an enumeration.
let weekend1 : Days = Days.Sat
let weekend2 : Days = Days.Sun
let weekDay1 : Days = Days.Mon

printfn "Monday: %A" weekDay1
printfn "Saturday: %A" weekend1
printfn "Sunday: %A" weekend2
```

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

```
Monday: Mon
Saturday: Sat
Sunday: Sun
```

F# - PATTERN MATCHING

Pattern matching allows you to “compare data with a logical structure or structures, decompose data into constituent parts, or extract information from data in various ways”.

In other terms, it provides a more flexible and powerful way of testing data against a series of conditions and performing some computations based on the condition met.

Conceptually, it is like a series of if... then statements.

Syntax

In high level terms, pattern matching follows this syntax in F# −

```
match expr with
```

```
| pat1 - result1
| pat2 -> result2
| pat3 when expr2 -> result3
| _ -> defaultResult
```

Where,

- Each | symbol defines a condition.
- The -> symbol means "if the condition is true, return this value...".
- The _ symbol provides the default pattern, meaning that it matches all other things like a wildcard.

Example 1

The following example, calculates the Fibonacci numbers using pattern matching syntax −

```
let rec fib n =
   match n with
   | 0 -> 0
   | 1 -> 1
   | _ -> fib (n - 1) + fib (n - 2)
for i = 1 to 10 do
   printfn "Fibonacci %d: %d" i (fib i)
```

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

```
Fibonacci 1: 1
Fibonacci 2: 1
Fibonacci 3: 2
Fibonacci 4: 3
Fibonacci 5: 5
Fibonacci 6: 8
Fibonacci 7: 13
Fibonacci 8: 21
Fibonacci 9: 34
Fibonacci 10: 55
```

You can also chain together multiple conditions, which return the same value. For example −

Example 2

```
let printSeason month =
   match month with
   | "December" | "January" | "February" -> printfn "Winter"
   | "March" | "April" -> printfn "Spring"
   | "May" | "June" -> printfn "Summer"
   | "July" | "August" -> printfn "Rainy"
   | "September" | "October" | "November" -> printfn "Autumn"
   | _ -> printfn "Season depends on month!"

printSeason "February"
printSeason "April"
printSeason "November"
printSeason "July"
```

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

```
Winter
Spring
Autumn
Rainy
```

Pattern Matching Functions

F# allows you to write pattern matching functions using the **function** keyword −

```
let getRate = function
   | "potato" -> 10.00
   | "brinjal" -> 20.50
   | "cauliflower" -> 21.00
   | "cabbage" -> 8.75
   | "carrot" -> 15.00
   | _ -> nan (* nan is a special value meaning "not a number" *)

printfn "%g"(getRate "potato")
printfn "%g"(getRate "brinjal")
printfn "%g"(getRate "cauliflower")
printfn "%g"(getRate "cabbage")
printfn "%g"(getRate "carrot")
```

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

```
10
20.5
21
8.75
15
```

Adding Filters or Guards to Patterns

You can add filters, or guards, to patterns using the **when** keyword.

Example 1

```
let sign = function
   | 0 -> 0
   | x when x < 0 -> -1
   | x when x > 0 -> 1

printfn "%d" (sign -20)
printfn "%d" (sign 20)
printfn "%d" (sign 0)
```

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

```
-1
1
0
```

Example 2

```
let compareInt x =
   match x with
   | (var1, var2) when var1 > var2 -> printfn "%d is greater than %d" var1 var2
   | (var1, var2) when var1 < var2 -> printfn "%d is less than %d" var1 var2
   | (var1, var2) -> printfn "%d equals %d" var1 var2

compareInt (11,25)
compareInt (72, 10)
compareInt (0, 0)
```

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

```
11 is less than 25
72 is greater than 10
0 equals 0
```

Pattern Matching with Tuples

The following example demonstrates the pattern matching with tuples −

```
let greeting (name, subject) =
   match (name, subject) with
   | ("Zara", _) -> "Hello, Zara"
   | (name, "English") -> "Hello, " + name + " from the department of English"
   | (name, _) when subject.StartsWith("Comp") -> "Hello, " + name + " from the
department of Computer Sc."
   | (_, "Accounts and Finance") -> "Welcome to the department of Accounts and Finance!"
   | _ -> "You are not registered into the system"

printfn "%s" (greeting ("Zara", "English"))
printfn "%s" (greeting ("Raman", "Computer Science"))
printfn "%s" (greeting ("Ravi", "Mathematics"))
```

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

```
Hello, Zara
Hello, Raman from the department of Computer Sc.
You are not registered into the system
```

Pattern Matching with Records

The following example demonstrates pattern matching with records −

```
type Point = { x: float; y: float }
let evaluatePoint (point: Point) =
   match point with
   | { x = 0.0; y = 0.0 } -> printfn "Point is at the origin."
   | { x = xVal; y = 0.0 } -> printfn "Point is on the x-axis. Value is %f." xVal
   | { x = 0.0; y = yVal } -> printfn "Point is on the y-axis. Value is %f." yVal
   | { x = xVal; y = yVal } -> printfn "Point is at (%f, %f)." xVal yVal

evaluatePoint { x = 0.0; y = 0.0 }
evaluatePoint { x = 10.0; y = 0.0 }
evaluatePoint { x = 0.0; y = 10.0 }
evaluatePoint { x = 10.0; y = 10.0 }
```

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

```
Point is at the origin.
Point is on the x-axis. Value is 10.000000.
Point is on the y-axis. Value is 10.000000.
Point is at (10.000000, 10.000000).
```

F# - EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. An F# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. F# exception handling provides the following constructs −

Construct	Description
raise expr	Raises the given exception.
failwith expr	Raises the **System.Exception** exception.
try expr with rules	Catches expressions matching the pattern rules.

try expr finally expr	Execution the **finally** expression both when the computation is successful and when an exception is raised.
| :? ArgumentException	A rule matching the given .NET exception type.
| :? ArgumentException as e	A rule matching the given .NET exception type, binding the name **e** to the exception object value.
| Failure(msg) → expr	A rule matching the given data-carrying F# exception.
| exn → expr	A rule matching any exception, binding the name **exn** to the exception object value.
| exn when expr → expr	A rule matching the exception under the given condition, binding the name **exn** to the exception object value.

Let us start with the basic syntax of Exception Handling.

Syntax

Basic syntax for F# exception handling block is −

```
exception exception-type of argument-type
```

Where,

- **exception-type** is the name of a new F# exception type.
- **argument-type** represents the type of an argument that can be supplied when you raise an exception of this type.
- Multiple arguments can be specified by using a tuple type for argument-type.

The **try...with** expression is used for exception handling in the F# language.

Syntax for the try ... with expression is −

```
try
   expression1
with
   | pattern1 -> expression2
   | pattern2 -> expression3
...
```

The **try...finally** expression allows you to execute clean-up code even if a block of code throws an exception.

Syntax for the try ... finally expression is −

```
try
   expression1
finally
   expression2
```

The **raise** function is used to indicate that an error or exceptional condition has occurred. It also captures the information about the error in an exception object.

Syntax for the raise function is −

```
raise (expression)
```

The **failwith** function generates an F# exception.

Syntax for the failwith function is −

```
failwith error-message-string
```

The **invalidArg** function generates an argument exception.

```
invalidArg parameter-name error-message-string
```

Example of Exception Handling

Example 1

The following program shows the basic exception handling with a simple try... with block −

```
let divisionprog x y =
   try
      Some (x / y)
   with
      | :? System.DivideByZeroException -> printfn "Division by zero!"; None

let result1 = divisionprog 100 0
```

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

```
Division by zero!
```

Example 2

F# provides an **exception** type for declaring exceptions. You can use an exception type directly in the filters in a **try...with** expression.

The following example demonstrates this −

```
exception Error1 of string
// Using a tuple type as the argument type.
exception Error2 of string * int

let myfunction x y =
   try
      if x = y then raise (Error1("Equal Number Error"))
      else raise (Error2("Error Not detected", 100))
   with
      | Error1(str) -> printfn "Error1 %s" str
      | Error2(str, i) -> printfn "Error2 %s %d" str i
myfunction 20 10
myfunction 5 5
```

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

```
Error2 Error Not detected 100
Error1 Equal Number Error
```

Example 3

The following example demonstrates nested exception handling −

```
exception InnerError of string
exception OuterError of string

let func1 x y =
   try
      try
         if x = y then raise (InnerError("inner error"))
         else raise (OuterError("outer error"))
```

```
      with
         | InnerError(str) -> printfn "Error:%s" str
   finally
      printfn "From the finally block."

let func2 x y =
   try
      func1 x y
   with
      | OuterError(str) -> printfn "Error: %s" str

func2 100 150
func2 100 100
func2 100 120
```

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

```
From the finally block.
Error: outer error
Error:inner error
From the finally block.
From the finally block.
Error: outer error
```

Example 4

The following function demonstrates the **failwith** function −

```
let divisionFunc x y =
   if (y = 0) then failwith "Divisor cannot be zero."
   else
      x / y

let trydivisionFunc x y =
   try
      divisionFunc x y
   with
      | Failure(msg) -> printfn "%s" msg; 0

let result1 = trydivisionFunc 100 0
let result2 = trydivisionFunc 100 4
printfn "%A" result1
printfn "%A" result2
```

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

```
Divisor cannot be zero.
0
25
```

Example 5

The **invalidArg** function generates an argument exception. The following program demonstrates this −

```
let days = [| "Sunday"; "Monday"; "Tuesday"; "Wednesday"; "Thursday"; "Friday";
"Saturday" |]
let findDay day =
   if (day > 7 || day < 1)
      then invalidArg "day" (sprintf "You have entered %d." day)
   days.[day - 1]

printfn "%s" (findDay 1)
printfn "%s" (findDay 5)
printfn "%s" (findDay 9)
```

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

```
Sunday
Thursday
Unhandled Exception:
System.ArgumentException: You have entered 9.
…
```

Some other information about the file and variable causing error in the system will also be displayed, depending upon the system.

F# - CLASSES

Classes are types that represent objects that can have properties, methods, and events. 'They are used to model actions, processes, and any conceptual entities in applications.'

Syntax

Syntax for defining a class type is as follows −

```
// Class definition:
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as
identifier ] =
   [ class ]
      [ inherit base-type-name(base-constructor-args) ]
      [ let-bindings ]
      [ do-bindings ]
      member-list
      ...
   [ end ]

// Mutually recursive class definitions:
type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

Where,

- The **type-name** is any valid identifier. Default access modifier for this is **public**.
- The **type-params** describes optional generic type parameters.
- The **parameter-list** describes constructor parameters. Default access modifier for primary constructor is **public**.
- The **identifier** used with the optional **as** keyword gives a name to the instance variable, or **self-identifier,** which can be used in the type definition to refer to the instance of the type.
- The **inherit** keyword allows you to specify the base class for a class.
- The **let** bindings allow you to declare fields or function values local to the class.
- The **do-bindings** section includes code to be executed upon object construction.
- The **member-list** consists of additional constructors, instance and static method declarations, interface declarations, abstract bindings, and property and event declarations.
- The keywords **class** and **end** that mark the start and end of the definition are optional.

Constructor of a Class

The constructor is code that creates an instance of the class type.

In F#, constructors work little differently than other .Net languages. In the class definition, the arguments of the primary constructor are described as parameter-list.

The body of the constructor consists of the **let** and **do** bindings.

You can add additional constructors by using the new keyword to add a member −

```
new (argument-list) = constructor-body
```

The following example illustrates the concept −

Example

The following program creates a line class along with a constructor that calculates the length of the line while an object of the class is created −

```
type Line = class
   val X1 : float
   val Y1 : float
   val X2 : float
   val Y2 : float

   new (x1, y1, x2, y2) as this =
      { X1 = x1; Y1 = y1; X2 = x2; Y2 = y2;}
      then
         printfn " Creating Line: {(%g, %g), (%g, %g)}\nLength: %g"
            this.X1 this.Y1 this.X2 this.Y2 this.Length

   member x.Length =
      let sqr x = x * x
      sqrt(sqr(x.X1 - x.X2) + sqr(x.Y1 - x.Y2) )
end
let aLine = new Line(1.0, 1.0, 4.0, 5.0)
```

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

```
Creating Line: {(1, 1), (4, 5)}
Length: 5
```

Let Bindings

The let bindings in a class definition allow you to define private fields and private functions for F# classes.

```
type Greetings(name) as gr =
   let data = name
   do
      gr.PrintMessage()
   member this.PrintMessage() =
      printf "Hello %s\n" data
let gtr = new Greetings("Zara")
```

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

```
Hello Zara
```

Please note the use of self-identifier *gr* for the *Greetings* class.

F# - STRUCTURES

A structure in F# is a value type data type. It helps you to make a single variable, hold related data of various data types. The **struct** keyword is used for creating a structure.

Syntax

Syntax for defining a structure is as follows −

```
[ attributes ]
type [accessibility-modifier] type-name =
   struct
      type-definition-elements
   end
// or
[ attributes ]
[<StructAttribute>]
type [accessibility-modifier] type-name =
   type-definition-elements
```

There are two syntaxes. The first syntax is mostly used, because, if you use the **struct** and **end** keywords, you can omit the **StructAttribute** attribute.

The structure definition elements provide −

- Member declarations and definitions.
- Constructors and mutable and immutable fields.
- Members and interface implementations.

Unlike classes, structures cannot be inherited and cannot contain let or do bindings. Since, structures do not have let bindings; you must declare fields in structures by using the **val** keyword.

When you define a field and its type using **val** keyword, you cannot initialize the field value, instead they are initialized to zero or null. So for a structure having an implicit constructor, the **val** declarations be annotated with the **DefaultValue** attribute.

Example

The following program creates a line structure along with a constructor. The program calculates the length of a line using the structure −

```
type Line = struct
   val X1 : float
   val Y1 : float
   val X2 : float
   val Y2 : float

   new (x1, y1, x2, y2) =
      {X1 = x1; Y1 = y1; X2 = x2; Y2 = y2;}
end
let calcLength(a : Line)=
   let sqr a = a * a
   sqrt(sqr(a.X1 - a.X2) + sqr(a.Y1 - a.Y2) )

let aLine = new Line(1.0, 1.0, 4.0, 5.0)
let length = calcLength aLine
printfn "Length of the Line: %g " length
```

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

```
Length of the Line: 5
```

F# - OPERATOR OVERLOADING

You can redefine or overload most of the built-in operators available in F#. Thus a programmer can use operators with user-defined types as well.

Operators are functions with special names, enclosed in brackets. They must be defined as static class members. Like any other function, an overloaded operator has a return type and a parameter list.

The following example, shows a + operator on complex numbers −

```
//overloading &plus; operator
static member (&plus;) (a : Complex, b: Complex) =
Complex(a.x &plus; b.x, a.y &plus; b.y)
```

The above function implements the addition operator (+) for a user-defined class Complex. It adds the attributes of two objects and returns the resultant Complex object.

Implementation of Operator Overloading

The following program shows the complete implementation −

```
//implementing a complex class with +, and - operators
//overloaded
type Complex(x: float, y : float) =
   member this.x = x
   member this.y = y
   //overloading + operator
   static member (+) (a : Complex, b: Complex) =
      Complex(a.x + b.x, a.y + b.y)

   //overloading - operator
   static member (-) (a : Complex, b: Complex) =
      Complex(a.x - b.x, a.y - b.y)

   // overriding the ToString method
   override this.ToString() =
      this.x.ToString() + " " + this.y.ToString()

//Creating two complex numbers
let c1 = Complex(7.0, 5.0)
let c2 = Complex(4.2, 3.1)

// addition and subtraction using the
//overloaded operators
let c3 = c1 + c2
let c4 = c1 - c2

//printing the complex numbers
printfn "%s" (c1.ToString())
printfn "%s" (c2.ToString())
printfn "%s" (c3.ToString())
printfn "%s" (c4.ToString())
```

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

```
7 5
4.2 3.1
11.2 8.1
2.8 1.9
```

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.
```

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 (upcast operator).

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
```

F# - EVENTS

Events allow classes to send and receive messages between one another.

In GUI, events are user actions like key press, clicks, mouse movements, etc., or some occurrence like system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

Objects communicate with one another through synchronous message passing.

Events are attached to other functions; objects register **callback** functions to an event, and these

callbacks are executed when (and if) the event is triggered by some object.

The Event Class and Event Module

The Control.Event<'T> Class helps in creating an observable object or event.

It has the following instance members to work with the events −

Member	Description
Publish	Publishes an observation as a first class value.
Trigger	Triggers an observation using the given parameters.

The Control.Event Module provides functions for managing event streams −

Value	Description
add : ('T → unit) → Event<'Del,'T> → unit	Runs the given function each time the given event is triggered.
choose : ('T → 'U option) → IEvent<'Del,'T> → IEvent<'U>	Returns a new event which fires on a selection of messages from the original event. The selection function takes an original message to an optional new message.
filter : ('T → bool) → IEvent<'Del,'T> → IEvent<'T>	Returns a new event that listens to the original event and triggers the resulting event only when the argument to the event passes the given function.
map : ('T → 'U) → IEvent<'Del, 'T> → IEvent<'U>	Returns a new event that passes values transformed by the given function.
merge : IEvent<'Del1,'T> → IEvent<'Del2,'T> → IEvent<'T>	Fires the output event when either of the input events fire.
pairwise : IEvent<'Del,'T> → IEvent<'T * 'T>	Returns a new event that triggers on the second and subsequent triggering of the input event. The **Nth** triggering of the input event passes the arguments from the **N-1th** and **Nth** triggering as a pair. The argument passed to the **N-1th** triggering is held in hidden internal state until the **Nth** triggering occurs.
partition : ('T → bool) → IEvent<'Del,'T> → IEvent<'T> * IEvent<'T>	Returns a new event that listens to the original event and triggers the first resulting event if the application of the predicate to the event arguments returned true, and the second event if it returned false.
scan : ('U → 'T → 'U) → 'U → IEvent<'Del,'T> → IEvent<'U>	Returns a new event consisting of the results of applying the given accumulating function to successive values triggered on the input event. An item of internal state records the current value of the state parameter. The internal state is not locked during the execution of the accumulation function, so care should be taken that the input IEvent not triggered by multiple threads simultaneously.
split : ('T → Choice<'U1,'U2>) → IEvent<'Del,'T> → IEvent<'U1> *	Returns a new event that listens to the original event and triggers the first resulting event if the

IEvent<'U2>	application of the function to the event arguments returned a Choice1Of2, and the second event if it returns a Choice2Of2.

Creating Events

Events are created and used through the **Event** class. The Event constructor is used for creating an event.

Example

```
type Worker(name : string, shift : string) =
   let mutable _name = name;
   let mutable _shift = shift;
   let nameChanged = new Event<unit>() (* creates event *)
   let shiftChanged = new Event<unit>() (* creates event *)

   member this.Name
      with get() = _name
      and set(value) = _name <- value

   member this.Shift
      with get() = _shift
      and set(value) = _shift <- value
```

After this you need to expose the nameChanged field as a public member, so that the listeners can hook onto the event for which, you use the **Publish** property of the event −

```
type Worker(name : string, shift : string) =
   let mutable _name = name;
   let mutable _shift = shift;

   let nameChanged = new Event<unit>() (* creates event *)
   let shiftChanged = new Event<unit>() (* creates event *)

   member this.NameChanged = nameChanged.Publish (* exposed event handler *)
   member this.ShiftChanged = shiftChanged.Publish (* exposed event handler *)

   member this.Name
      with get() = _name
      and set(value) = _name <- value
      nameChanged.Trigger() (* invokes event handler *)

   member this.Shift
      with get() = _shift
      and set(value) = _shift <- value
   shiftChanged.Trigger() (* invokes event handler *)
```

Next, you add callbacks to event handlers. Each event handler has the type IEvent<'T>, which provides several methods −

Method	Description
val Add : event:('T → unit) → unit	Connects a listener function to the event. The listener will be invoked when the event is fired.
val AddHandler : 'del → unit	Connects a handler delegate object to the event. A handler can be later removed using RemoveHandler. The listener will be invoked when the event is fired.
val RemoveHandler : 'del → unit	Removes a listener delegate from an event listener store.

The following section provides a complete example.

Example

The following example demonstrates the concept and techniques discussed above −

```
type Worker(name : string, shift : string) =
   let mutable _name = name;
   let mutable _shift = shift;

   let nameChanged = new Event<unit>() (* creates event *)
   let shiftChanged = new Event<unit>() (* creates event *)

   member this.NameChanged = nameChanged.Publish (* exposed event handler *)
   member this.ShiftChanged = shiftChanged.Publish (* exposed event handler *)

   member this.Name
      with get() = _name
      and set(value) =
         _name <- value
         nameChanged.Trigger() (* invokes event handler *)

   member this.Shift
      with get() = _shift
      and set(value) =
         _shift <- value
         shiftChanged.Trigger() (* invokes event handler *)

let wk = new Worker("Wilson", "Evening")
wk.NameChanged.Add(fun () -> printfn "Worker changed name! New name: %s" wk.Name)
wk.Name <- "William"
wk.NameChanged.Add(fun () -> printfn "-- Another handler attached to NameChanged!")
wk.Name <- "Bill"

wk.ShiftChanged.Add(fun () -> printfn "Worker changed shift! New shift: %s" wk.Shift)
wk.Shift <- "Morning"
wk.ShiftChanged.Add(fun () -> printfn "-- Another handler attached to ShiftChanged!")
wk.Shift <- "Night"
```

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

```
Worker changed name! New name: William
Worker changed name! New name: Bill
-- Another handler attached to NameChanged!
Worker changed shift! New shift: Morning
Worker changed shift! New shift: Night
-- Another handler attached to ShiftChanged!
```

F# - MODULES

As per MSDN library, an F# module is a grouping of F# code constructs, such as types, values, function values, and code in do bindings. It is implemented as a common language runtime (CLR) class that has only static members.

Depending upon the situation whether the whole file is included in the module, there are two types of module declarations −

- Top-level module declaration
- Local module declaration

In a top-level module declaration the whole file is included in the module. In this case, the first declaration in the file is the module declaration. You do not have to indent declarations in a top-level module.

In a local module declaration, only the declarations that are indented under that module declaration are part of the module.

Syntax

Syntax for module declaration is as follows −

```
// Top-level module declaration.
module [accessibility-modifier] [qualified-namespace.]module-name
   declarations
// Local module declaration.
module [accessibility-modifier] module-name =
   declarations
```

Please note that the accessibility-modifier can be one of the following − public, private, internal. The default is **public**.

The following examples will demonstrate the concepts −

Example 1

The module file Arithmetic.fs −

```
module Arithmetic
let add x y =
   x + y

let sub x y =
   x - y

let mult x y =
   x * y

let div x y =
   x / y
```

The program file main.fs −

```
// Fully qualify the function name.
let addRes = Arithmetic.add 25 9
let subRes = Arithmetic.sub 25 9
let multRes = Arithmetic.mult 25 9
let divRes = Arithmetic.div 25 9

printfn "%d" addRes
printfn "%d" subRes
printfn "%d" multRes
printfn "%d" divRes

// Opening the module.
open Arithmetic

let addRes2 = Arithmetic.add 100 10
let subRes2 = Arithmetic.sub 100 10
let multRes2 = Arithmetic.mult 100 10
let divRes2 = Arithmetic.div 100 10

printfn "%d" addRes2
printfn "%d" subRes2
printfn "%d" multRes2
printfn "%d" divRes2
```

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

```
34
16
225
2
110
```

```
90
1000
10
```

Example 2

```
// Module1
module module1 =
   // Indent all program elements within modules that are declared with an equal sign.
   let value1 = 100
   let module1Function x =
      x + value1

// Module2
module module2 =
   let value2 = 200

   // Use a qualified name to access the function.
   // from module1.
   let module2Function x =
      x + (module1.module1Function value2)

let result = module1.module1Function 25
printfn "%d" result

let result2 = module2.module2Function 25
printfn "%d" result2
```

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

```
125
325
```

F# - NAMESPACES

A **namespace** is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace will not conflict with the same class names declared in another.

As per the MSDN library, a **namespace** lets you organize code into areas of related functionality by enabling you to attach a name to a grouping of program elements.

Declaring a Namespace

To organize your code in a namespace, you must declare the namespace as the first declaration in the file. The contents of the entire file then become part of the namespace.

```
namespace [parent-namespaces.]identifier
```

The following example illustrates the concept −

Example

```
namespace testing

module testmodule1 =
   let testFunction x y =
      printfn "Values from Module1: %A %A" x y
module testmodule2 =
   let testFunction x y =
      printfn "Values from Module2: %A %A" x y

module usermodule =
   do
      testmodule1.testFunction ( "one", "two", "three" ) 150
```

```
    testmodule2.testFunction (seq { for i in 1 .. 10 do yield i * i }) 200
```

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

```
Values from Module1: ("one", "two", "three") 150
Values from Module2: seq [1; 4; 9; 16; ...] 200
```