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.