LISP - ERROR HANDLING

In Common LISP terminology, exceptions are called conditions.

In fact, conditions are more general than exceptions in traditional programming languages, because a **condition** represents any occurrence, error, or not, which might affect various levels of function call stack.

Condition handling mechanism in LISP, handles such situations in such a way that conditions are used to signal warning *saybyprintinganwarning* while the upper level code on the call stack can continue its work.

The condition handling system in LISP has three parts:

- Signalling a condition
- Handling the condition
- Restart the process

Handling a Condition

Let us take up an example of handling a condition arising out of divide by zero condition, to explain the concepts here.

You need to take the following steps for handling a condition:

- **Define the Condition** - "A condition is an object whose class indicates the general nature of the condition and whose instance data carries information about the details of the particular circumstances that lead to the condition being signalled".

  The define-condition macro is used for defining a condition, which has the following syntax:

```
(define-condition condition-name (error)
   ((text :initarg :text :reader text))
)
```

  New condition objects are created with MAKE-CONDITION macro, which initializes the slots of the new condition based on the **:initargs** argument.

  In our example, the following code defines the condition:

```
(define-condition on-division-by-zero (error)
   ((message :initarg :message :reader message))
)
```

- **Writing the Handlers** - a condition handler is a code that are used for handling the condition signalled thereon. It is generally written in one of the higher level functions that call the erroring function. When a condition is signalled, the signalling mechanism searches for an appropriate handler based on the condition's class.

  Each handler consists of:

  - Type specifier, that indicates the type of condition it can handle
  - A function that takes a single argument, the condition

  When a condition is signalled, the signalling mechanism finds the most recently established handler that is compatible with the condition type and calls its function.

  The macro **handler-case** establishes a condition handler. The basic form of a handler-case:

```
(handler-case expression error-clause*)
```

Where, each error clause is of the form:

```
condition-type ([var]) code)
```

- **Restarting Phase**

  This is the code that actually recovers your program from errors, and condition handlers can then handle a condition by invoking an appropriate restart. The restart code is generally place in middle-level or low-level functions and the condition handlers are placed into the upper levels of the application.

  The **handler-bind** macro allows you to provide a restart function, and allows you to continue at the lower level functions without unwinding the function call stack. In other words, the flow of control will still be in the lower level function.

  The basic form of **handler-bind** is as follows:

  ```
  (handler-bind (binding*) form*)
  ```

  Where each binding is a list of the following:

  - a condition type
  - a handler function of one argument

  The **invoke-restart** macro finds and invokes the most recently bound restart function with the specified name as argument.

  You can have multiple restarts.

Example

In this example, we demonstrate the above concepts by writing a function named division-function, which will create an error condition if the divisor argument is zero. We have three anonymous functions that provide three ways to come out of it - by returning a value 1, by sending a divisor 2 and recalculating, or by returning 1.

Create a new source code file named main.lisp and type the following code in it.

```
(define-condition on-division-by-zero (error)
   ((message :initarg :message :reader message))
)

(defun handle-infinity ()
   (restart-case
      (let ((result 0))
         (setf result (division-function 10 0))
         (format t "Value: ~a~%" result)
      )
      (just-continue () nil)
   )
)

(defun division-function (value1 value2)
   (restart-case
      (if (/= value2 0)
         (/ value1 value2)
         (error 'on-division-by-zero :message "denominator is zero")
      )

      (return-zero () 0)
      (return-value (r) r)
      (recalc-using (d) (division-function value1 d))
   )
)
```

```
(defun high-level-code ()
   (handler-bind
      (
         (on-division-by-zero
            #'(lambda (c)
               (format t "error signaled: ~a~%" (message c))
               (invoke-restart 'return-zero)
            )
         )
         (handle-infinity)
      )
   )
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'return-value 1)
         )
      )
   )
   (handle-infinity)
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'recalc-using 2)
         )
      )
   )
   (handle-infinity)
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'just-continue)
         )
      )
   )
   (handle-infinity)
)

(format t "Done."))
```

When you execute the code, it returns the following result:

```
error signaled: denominator is zero
Value: 1
error signaled: denominator is zero
Value: 5
error signaled: denominator is zero
Done.
```

Apart from the 'Condition System', as discussed above, Common LISP also provides various functions that may be called for signalling an error. Handling of an error, when signalled, is however, implementation-dependent.

Error Signalling Functions in LISP

The following table provides commonly used functions signalling warnings, breaks, non-fatal and

fatal errors.

The user program specifies an error message *astring*. The functions process this message and may/may not display it to the user.

The error messages should be constructed by applying the **format** function, should not contain a newline character at either the beginning or end, and need not indicate error, as the LISP system will take care of these according to its preferred style.

SL No.	Functions and Descriptions
1	**error** *format-string* &rest *args* It signals a fatal error. It is impossible to continue from this kind of error; thus error will never return to its caller.
2	**cerror** *continue-format-string error-format-string* &rest *args* It signals an error and enters the debugger. However, it allows the program to be continued from the debugger after resolving the error.
3	**warn** *format-string* &rest *args* it prints an error message but normally doesn't go into the debugger
4	**break** &optional *format-string* &rest *args* It prints the message and goes directly into the debugger, without allowing any possibility of interception by programmed error-handling facilities

Example

In this example, the factorial function calculates factorial of a number; however, if the argument is negative, it raises an error condition.

Create a new source code file named main.lisp and type the following code in it.

```
(defun factorial (x)
   (cond ((or (not (typep x 'integer)) (minusp x))
      (error "~S is a negative number." x))
      ((zerop x) 1)
      (t (* x (factorial (- x 1))))
   )
)

(write(factorial 5))
(terpri)
(write(factorial -1))
```

When you execute the code, it returns the following result:

```
120
*** - -1 is a negative number
```