Lemick Language Reference

Modification date: 04.04.2004

This documentation is incomplete and is under development now

Contents

Chapter 1

Language Core

1.1 General Information

1.2 Syntax

1.2.1 Comments

1.2.2 Constants

1.2.3 Expressions and Operators

1.2.4 Standard Types

Lemick defines the following types: *Byte*, *Integer*, *Long*, *Single*, *Double*, *String* and *Object*.

Byte is a signed 8-bit value. Its range is from -128 to +127. Byte type is useful for storing and manipulating arbitrary data.

Integer is a signed 32-bit value. Its range is from -2147483648 to +2147483647.

Long is a signed 64-bit value. Its range is from -9223372036854775808 to +9223372036854775807.

Single is a 32-bit IEEE-754 single precision type. Its range is from -3.4e-38 to 3.4e+38.

Double is a 64-bit IEEE-754 double precision type. Its range is from -1.7e-308 to 1.7e+308.

String is a Lemick built-in type for manipulation on string values. Though this type is declared as a part of the language its structure is not specified and operations on this type are not provided in the language core.
Standard package lemick.strings provides implementation of this type, provides overloaded "+" operator for strings concatenation, conversions from and to other types and etc.

Object is the the super-type for all the user types declared with type or Class keyword. All such types can be down-casted to Object type.

1.2.5 Type casting

1.3 Keywords

1.3.1 Abstract

Abstract, a class modifier

Syntax:
```
ABSTRACT CLASS Foo
```

Abstract says that it is prohibited to create instances of the declared class though there may be no other causes preventing this. As a class modifier, *abstract* can not be used in conjunction with with *Final* modifier.

Abstract, a method modifier

Syntax:
```
DECLARE ABSTRACT SUB Foo
```

In a normal Lemick class any method prototyped with *Declare* statement is an abstract method thus causing the containing class to became abstract. For an extern class the prototyped methods are considered as normal with their body declared somewhere else. To define the particular method in an extern class as an abstract you must place *Abstract* keyword just after *Declare*.
For the classes extending a abstract class with the abstract methods the compiler will require implementation of all the abstract methods unless the declared class is abstract itself.

1.3.2 And

Syntax:
```
a AND b
```

And is logical and operator. It evaluates to non-zero (truth value) when both its operands are non-zero.
Operator *And* can be applied only to integer types, such as Byte, Integer and Long.

1.3.3 As

Syntax:
```
DIM Var AS TypeName
Member As TypeName
SUB Foo(Argument AS TypeName)
FUNCTION Foo AS TypeName
```

Keyword *As* precedes type name in declaration of variables, formal arguments,

type members as function return values. It is not an operator or a statement, it is merely part of the syntax which make Lemick grammar easier to parse and read.

See also: Dim (1.3.12), Class (1.3.7), Sub (1.3.51), Function (1.3.24)

1.3.4 Case

Syntax:
`CASE = value` : *statements block*
`CASE IS binary_op value` : *statements block*
`CASE IS value` : *statements block*
`CASE ranges` : *statements block*

Statement *Case* is the part of *Select Case* construct - a powerful alternative to switch-like constructs in other languages. *Case* has four different forms as given in the syntax section. Each form has condition and block of statements that are executed if the condition holds. Independently of the result of the previous *Case* block, execution always proceeds with the next *Case*, unless the shole Select Case is aborted using *Exit Select* statement.
In the first syntax case the control value (specified in Select Case) is matched against test `value`.
Equality test operator can be substituted with any other binary ordering operator. In the second syntax, `binary_op` is one of the following: `=, >, <, <>, <=, >=`.
In the third syntax, `value` is any arbitrary expression that may or may not contain the control value.
And in the the last *Case* form you can use list of test expressions and ranges. Syntax of `ranges` is following:
ranges := range | ranges , range;
range := expression to expression | expression.

Ranges with *To* keyword give the lower and the upper bound for the control value. Corresponding operations are less or equal for the lower bound and greater or equal for the upper bound.

Example: `case 2, 7 to 17, 23 to 29, 97, q > 3`

Note that it is possible to use *Case* constructs with non-numeric types and any arbitrary types if the corresponding operators are overloaded for the used types.

See also: Select Case (1.3.44), Case Else (1.3.5)

1.3.5 Case Else

Syntax:
`CASE ELSE:` *statements block*

To provide special action in the case when no *Case* blocks executed, you can

place *Case Else* statement *at the bottom* of the corresponding *Select Case* construct.

See also: Select Case (1.3.44), Case (1.3.4)

1.3.6 Catch

Syntax:
`CATCH var AS ExceptionClass:`*statements block*

Catch block contains code for recovery from the exception *ExceptionClass*. It is a part of *Try-Catch-Finally* construct.

1.3.7 Class

Syntax:
`CLASS Foo [EXTENDS Bar] [IMPLEMENTS iface1, iface2, ...]:`*declarations*`:END CLASS`

See also: Type (1.3.54), Interface (1.3.29)

1.3.8 Concurrent

Syntax:
```
Foo [arg1, arg2, ...]  CONCURRENT [ThreadHandle]
```

1.3.9 Const

Syntax:
```
CONST [SHARED] Var = value [AS TypeName]
```

See also: Dim (1.3.12)

1.3.10 Continue

Syntax:
```
CONTINUE [LabelName]
```

See also: Exit (1.3.18), Repeat (1.3.42)

1.3.11 Declare

Syntax:
```
DECLARE SUB Foo [(formal arguments)]
DECLARE FUNCTION Foo [(formal arguments)] [AS ReturnType]
DECLARE CLASS Foo
```

```
DECLARE INTERFACE Foo
```

See also: Sub (1.3.51), Function (1.3.24), Class (1.3.7), Interface (1.3.29)

1.3.12 Dim

Syntax:
```
DIM [SHARED] Var1 [AS TypeName], Var2 [AS TypeName], ...
DIM AS [SHARED] Var1, Var2, ...
```

See also: Public (1.3.40), Private (1.3.38), Shared (1.3.46), Const (1.3.9)

1.3.13 Do

Syntax:
```
DO WHILE condition [LabelName]:  ...  :  LOOP
DO [LabelName]:  ...  :  LOOP WHILE condition
DO [LabelName]:  ...  :  LOOP UNTIL condition
```

See also: While (1.3.58), Exit Loop (1.3.18)

1.3.14 Down To

Syntax:
```
FOR i = ubound DOWN TO lbound [STEP step]
```
Note:
Down To may be also written without a space, i.e. *DownTo*
See also: For (1.3.23)

1.3.15 Else

Syntax:
```
IF condition THEN then_statements ELSE else_statements
```

See also: For (1.3.26)

1.3.16 End

Syntax:
```
END
```

1.3.17 End ...

Syntax:
```
END INTERFACE
END CLASS
END TYPE
END PROPERTY
```

```
END FUNCTION
END SUB
END SELECT
END IF
```

1.3.18 Exit

Syntax:

```
EXIT FUNCTION
EXIT SUB
EXIT FOR
EXIT LOOP
EXIT SELECT
```

1.3.19 Extends

Syntax:

```
CLASS Foo EXTENDS Bar
INTERFACE Foo EXTENDS Bar
```

1.3.20 Extern

Syntax:

```
CLASS Foo EXTERN
DECLARE SUB Foo EXTERN
```

1.3.21 Finally

Syntax:

```
FINALLY: ...
```

1.3.22 Final

Syntax:

```
FINAL CLASS Foo
FINAL SUB Foo
```

1.3.23 For

Syntax:
```
FOR i = value1 [DOWN] TO value2 [STEP step]:  ...  :NEXT
```

1.3.24 Function

Syntax:
```
FUNCTION Foo [(formal arguments)] [AS ReturnType]:  ...  :END FUNCTION
DECLARE FUNCTION Foo [(formal arguments)] [AS ReturnType]
```

1.3.25 Get

Syntax:
```
PROPERTY GET Name AS TypeName:  ...  :END PROPERTY
DECLARE PROPERTY GET Name AS TypeName
```

1.3.26 If

Syntax:
```
IF condition THEN then_statements
IF condition THEN then_statements ELSE else_statements
IF condition THEN then_block END IF
IF condition THEN then_block ELSE else_block END IF
```

1.3.27 Implements

Syntax:
```
CLASS Foo IMPLEMENTS iface1, iface2, ...
```

1.3.28 ImpLib

Syntax:
```
IMPLIB libname
```

1.3.29 Interface

Syntax:
INTERFACE Foo [EXTENDS Iface]:*declarations*:END INTERFACE

```
DECLARE INTERFACE Foo [EXTENDS Iface]
```

1.3.30 Is

Syntax:
`expression IS TypeName`
`CASE IS binary_op value:` *statements_block*
`CASE IS value:` *statements_block*

1.3.31 Loop

Syntax:
```
DO WHILE condition [LabelName]:  ...  :LOOP
DO [LabelName]:  ...  :LOOP WHILE condition
DO [LabelName]:  ...  :LOOP UNTIL condition
```

1.3.32 Mod

Syntax:
```
a MOD b
```

MOD is modulus arithmetic operator. It returns the value that is the remainder of an integer division of its arguments. Operator *MOD* can be applied only to integer types, such as Byte, Integer and Long.

1.3.33 New

Syntax:
```
NEW TypeName [(constructor arguments)]
NEW(array dimensions) AS TypeName
= NEW TypeName [(constructor arguments)]
```

1.3.34 Next

Syntax:
```
NEXT
```

See also: For (1.3.23)

1.3.35 Not

Syntax:
```
NOT a
```

NOT is logical negation operator. It returns non-zero (truth) value if its argument is zero (false).
Operator *NOT* can be used with any types, including integer and float numeric types, String, Object and any user types. For numeric types *NOT* holds if the argument contains zero value in the respective format, for String type *NOT* holds if the argument does not contain any string or is undefined. For Object and user types *NOT* holds if the respective argument does not point at any valid object instance.

1.3.36 Or

Syntax:
```
a OR b
```

And is logical *or* operator. It evaluates to non-zero (truth value) when both its operands are non-zero.
Operator *Or* can be applied only to integer types, such as Byte, Integer and Long.

See also: And (1.3.2), Not(1.3.35)

1.3.37 Package

Syntax:
```
PACKAGE PackageName
```

Declares the current program as a package description and implementation. Lemick compiler will place in the generate byte code additional information about the exported routines, classes, constants and variables.

1.3.38 Private

Syntax:
```
PRIVATE CLASS Foo
PRIVATE INTERFACE Foo
PRIVATE SUB Foo
PRIVATE CONST Var1 [AS TypeName] = value, ...
PRIVATE STATIC Var1 [AS TypeName], Var2 [AS TypeName], ...
PRIVATE Var1 [AS TypeName], Var2 [AS TypeName], ...
```

1.3.39 Property

Syntax:

```
PROPERTY GET Name AS TypeName:  ...  :END PROPERTY
PROPERTY SET Name(value AS TypeName):  ...  :END PROPERTY
DECLARE PROPERTY GET Name AS TypeName
DECLARE PROPERTY SET Name(value AS TypeName)
```

1.3.40 Public

Syntax:

```
PUBLIC CLASS Foo
PUBLIC INTERFACE Foo
PUBLIC SUB Foo
PUBLIC CONST Var1 [AS TypeName] = value, ...
PUBLIC STATIC Var1 [AS TypeName], Var2 [AS TypeName], ...
PUBLIC Var1 [AS TypeName], Var2 [AS TypeName], ...
```

1.3.41 Raise

Syntax:

```
RAISE expression
RAISE
```

1.3.42 Repeat

Syntax:

```
REPEAT [LabelName]
```

1.3.43 Retry

Syntax:

```
RETRY
```

This is a reserved keyword. It is not used in the current version.

1.3.44 Select Case

Syntax:

SELECT CASE ControlValue: *Case blocks* :END SELECT

1.3.45 Set

Syntax:
```
PROPERTY SET Name(value AS TypeName):  ...  :END PROPERTY
DECLARE PROPERTY SET Name(value AS TypeName)
```

1.3.46 Shared

Syntax:
```
DIM [SHARED] Var1 [AS TypeName], Var2 [AS TypeName], ...
DIM AS [SHARED] Var1, Var2, ...
CONST [SHARED] Var = value [AS TypeName]
```

1.3.47 Signal

Syntax:
```
SIGNAL expression
SIGNAL
```

1.3.48 Signals

Syntax:
```
CLASS Foo SIGNALS Exception1, Exception2, ...
SUB Foo SIGNALS Exception1, Exception2, ...
FUNCTION Foo SIGNALS Exception1, Exception2, ...
```

1.3.49 Static

Syntax:
```
STATIC SUB Foo
STATIC FUNCTION Foo
STATIC Var1 [AS TypeName], Var2 [AS TypeName], ...
```

1.3.50 Step

Syntax:
```
FOR i = value1 [DOWN] TO value2 [STEP step]:  ...  :  NEXT
```

1.3.51 Sub

Syntax:
```
SUB Foo [(formal arguments)]:  ...  :END SUB
DECLARE SUB Foo [(formal arguments)]
```

1.3.52 Then

Syntax:
```
IF condition THEN then_statements
IF condition THEN then_statements ELSE else_statements
IF condition THEN then_block END IF
IF condition THEN then_block ELSE else_block END IF
```

Then keyword is used only in context of *If* statement.

1.3.53 To

Syntax:
FOR i = value1 [DOWN] TO value2 [STEP step]: ... :NEXT
CASE ranges :*statements block*

1.3.54 Type

Syntax:
TYPE Foo [EXTENDS Bar]:*declarations*:END TYPE

1.3.55 Until

Syntax:
```
DO [LabelName]:  ...  :LOOP UNTIL condition
```

1.3.56 Use

Syntax:
```
USE Package1 [, Package2, ...]
```

1.3.57 Wend

Syntax:
```
WHILE condition [LabelName]:  ...  :WEND
```

1.3.58 While

Syntax:

```
WHILE condition [LabelName]:  ...  :WEND
DO [LabelName]:  ...  :LOOP WHILE condition
```

1.3.59 XOr

Syntax:

```
a XOR b
```

XOr is bitwise *exclusive or* operator. Operator *XOr* can be applied only to integer types, such as Byte, Integer and Long.