# Simple Language Specification

Draft version
August 2009

Author: Herbert Czymontek

# Grammar

The syntax is specified using Extended Backus-Naur Form (EBNF). The following symbols are used:

- | - separates alternatives
- () - defines groups
- [] - specifies an option (0 or 1 times)
- {} - specifies repetition (0 to n times)

Lexical symbols are highlighted in bold and blue, e.g. the digit one: **1**.

# Lexical Structure

Source files are plain text files.
The default character encoding is UTF-8.
Unlike other BASIC dialects, Simple is case-sensitive.

## Line terminators

Simple is a line-based language. That means that statements are separated by line terminators. The following line terminators are recognized:

- the Unicode character \u000D (CR - carriage return)
- the Unicode character \u000A (LF - line feed)
- the Unicode character \u000D (CR - carriage return) followed by the Unicode character \u000A (LF - line feed)

```
EndOfLine := (CR | LF | CR LF)
```

Line terminators can be escaped by using a single line continuation character (Unicode character \u005F (_) immediately before a line terminator (note that _ is not a legal character to start an identifier).

## Whitespace characters

The following characters are considered whitespace characters and can be used to separate tokens which would otherwise interpreted as a single token:

- the Unicode character \u0009 (TAB - tab)
- the Unicode character \u000B (VT - vertical tab)

- the Unicode character \u000C (FF - form feed)
- the Unicode character \u0020 (SP - space)

Whitespace := (TAB | VT | FF | SP )

## Tokens

The input character stream of a source file is reduced into a sequence of tokens. Tokens are the terminal symbols of the grammar.
The tokenizer will always attempt to find the longest possible character sequence that defines a token. For example the input character sequence "a<<b" will be tokenized as identifer a, left shift operator << and identifer b, while "a< <b would be tokenized as identifier a, less operator, less operator and identifier b.

## Comments

Comments are lexically treated like whitespace characters. Simple currently only supports single line comments. They start with the Unicode character \u0027 (') and end with the following line terminator. Comments cannot start within a string literal.

Comment := ' { any Unicode character except EndOfLine } EndOfLine

## Keywords

The following character sequences are reserved as keywords and cannot be used for identifiers:

Keywords := **Alias** | **And** | **As** | **Boolean** | **ByRef** | **Byte** | **ByVal** | **Case** | **Const** | **Date** | **Dim** | **Do** | **Double** | **Each** | **Else** | **ElseIf** | **End** | **Error** | **Event** | **Exit** | **For** | **Function** | **Get** | **If** | **In** | **Integer** | **Is** | **IsNot** | **Like** | **Long** | **Me** | **Mod** | **Next** | **New** | **Not** | **Nothing** | **Object** | **Or** | **On** | **Property** | **RaiseEvent** | **Select** | **Set** | **Short** | **Single** | **Static** | **Step** | **String** | **Sub** | **Then** | **To** | **TypeOf** | **Until** | **Variant** | **While** | **Xor**

## Identifiers

An identifier is a character sequence starting with a Java™ letter character, followed by Java™ letters or Java™ digit characters or the Unicode character \u005F (_). A Java™ letter character is defined as a character for which the Java™ method `java.lang.Character.JavaLetter()` returns true. A Java™ digit character is similarly defined as a character for which the Java™ method `java.lang.Character.JavaDigit()` returns true.

An identifier must not have the spelling as a keyword or literal.

Identifier := JavaLetter { JavaLetter | JavaDigit | **_** }

## Literals

A literal defines a constant typed value. Simple has integer literals, floating point literals, boolean literals and string literals, as well as a special object literal.

Literal := IntegerLiteral | FloatingPointLiteral | BooleanLiteral | StringLiteral | ObjectLiteral

### Integer Literals

Integer literals are numerical constants that fit without loss of precision into either the Integer type or the Long type. They can either be given in either decimal or hexadecimal format. Integer literals are always unsigned. To express negative constants, an integer literal must be prefixed by the unaray minus operator (**-**).

IntegerLiteral := DecimalIntegerLiteral | HexIntegerLiteral
DecimalIntegerLiteral := **0** | NonZeroDigit { Digit }
NonZeroDigit := **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
Digit := **0** | NonZeroDigit
HexIntegerLiteral := **&H** HexDigit { HexDigit }
HexDigit := Digit | **A** | **B** | **C** | **D** | **E** | **F**

### Floating Point Literals

Floating point literals are numerical constants that fit without loss of precision into either the Single type or the Double type. Floating point literals are always unsigned. To express negative constants, a floating point literal must be prefixed by the unaray minus operator (**-**).

FloatingPointLiteral := DecimalIntegerLiteral **.** Digit { Digit } [ Exponent ]
Exponent := **E** [ **+** | **-** ] DecimalIntegerLiteral

### Boolean Literals

Boolean literals are of the Boolean type and can assume one of the two values, true or false.

BooleanLiteral := **True** | **False**

### String Literals

A string literal is a sequence of Unicode characters enclosed by quotation marks (**"**). It is always of the String type. In order to use a quotation mark within the string literal, it must be escaped by preceeding it with a backslash character (****). To use a backslash character within the string literal it must be preceeded by another backslash character. Other characters that can be escaped are the newline character (**\n**), the carriage return character (**\r**), the tab character (**\t**) and the form feed character (**\f**).

StringLiteral := **"** { StringCharacter } **"**
StringCharacter := InputCharacter except **"** and **** | EscapedCharacter
InputCharacter := any 16-bit Unicode character
EscapedCharacter := **** | **\"** | **\n** | **\r** | **\t** | **\f**

### Object Literal

The special **Nothing** literal is used to initialize Object type references with no object.

ObjectLiteral := **Nothing**

## Separators

The following is a list of separator tokens used in Simple:

Separator := **(** | **)** | **,** | **:**

### Operators

The following is a list of operator tokens used in Simple expressions:

Operator := **<<** | **<** | **<=** | **=** | **<>** | **>=** | **>** | **>>** | **&** | **+** | **-** | **** | ***** | **/** | **^** | **.**

## Types and Values

### Primitive Types

Type := NonArrayType | ArrayType
NonArrayType := **Boolean** | **Byte** | **Short** | **Integer** | **Long** | **Single** | **Double** | **String** | **Date** | **Variant** | ObjectType

#### Boolean Type

The Boolean type is used to represent the two logical values *true* and *false*. Simple defines the corresponding literals **True** and **False** for these values.

#### Byte Type

The Byte type is a numerical type to represent 8-bit signed integer values in the range from -128 to 127.

#### Short Type

The Short type is a numerical type to represent 16-bit signed integer values in the range from -32768 to 32767.

#### Integer Type

The Integer type is a numerical type to represent 32-bit signed integer values in the range from -2147483648 to 2147483647.

### Long Type

The Long type is a numerical type to represent 64-bit signed integer values in the range from -9223372036854775808 to 9223372036854775807.

### Single Type

The Single type is a numerical type to represent single-precision 32-bit format IEEE 754 floating point values as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

### Double Type

The Double type is a numerical type to represent double-precision 64-bit format IEEE 754 floating point values as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

### String Type

The String type represents a possibly empty sequence of characters.

### Date Type

The Date type maps to *`java.util.Calendar`* class defined by Java™.

### Variant Type

The Variant type is a disjoint union able to represent any of the other data types (primitive types as well as array and object types).

## Array Types

ArrayType := NonArrayType **(** [ Expression { **,** Expression } ] | { **,** } **)**

Array types represent elements of the same type for access by subscript. Arrays can have one or more dimensions (up to 256).

The Array type may either specify the maximum number of elements for each dimension or leave the maximum number of elements unspecified (dynamically size arrays).

## Object Types

ObjectType := **Object** | Identifier

The values for the Object Type are references to object instances or the special value indicating no object instance, **Nothing**. Inheritance is treated as a design time property, meaning that there is no explicit syntax to specify object relationships. Instead this information is stored in a special section of the source file (see appendix for more information about the source file format). Simple supports both implementation as well as interface inheritance. An interface object may only define constants and bodyless functions and procedures.
There are two predefined events for (non-interface) objects:

- The Load event will be raised upon the first reference of any member of the object.
- The Initialize event will be raised upon allocation of an object instance (as part of the execution of the **New** operator).

## Default values

The following table shows the default values for the types. Variables are implicitly assigned the default variable corresponding to their type.

| **Boolean** | **Byte** | **Short** | **Integer** | **Long** | **Single** | **Double** | **String** | **Date** | **Variant** | **Array** | **Object** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| False | 0 | 0 | 0 | 0 | 0.0 | 0.0 | "" | Nothing | *any default Value* | Nothing | Nothing |

## Type Conversions

The following table shows the effect of conversions on values from one type to another.

| From/To | **Boolean** | **Byte** | **Short** | **Integer** | **Long** | **Single** | **Double** | **String** | **Date** | **Variant** | **Array** | **Object** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Boolean** | | True -> -1<br>False -> 0 | True -> -1<br>False -> 0 | True -> -1<br>False -> 0 | True -> -1<br>False -> 0 | True -> -1.0<br>False -> 0.0 | True -> -1.0<br>False -> 0.0 | True -> "True"<br>False -> "False" | Compile time error | | Compile time error | Compile time error |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Byte** | 0 -> False other values-> True | | | | | | | Integer value in decimal format | Compile time error | | Compile time error | Compile time error |
| **Short** | 0 -> False other values-> True | Possible loss of significant digits | | | | | | Integer value in decimal format | Compile time error | | Compile time error | Compile time error |
| **Integer** | 0 -> False other values-> True | Possible loss of significant digits | Possible loss of significant digits | | | Possible loss of precision | | Integer value in decimal format | Compile time error | | Compile time error | Compile time error |
| **Long** | 0 -> False other values-> True | Possible loss of significant digits | Possible loss of significant digits | Possible loss of significant digits | | Possible loss of precision | Possible loss of precision | Integer value in decimal format | Compile time error | | Compile time error | Compile time error |
| **Single** | 0 -> False other values-> True | Possible loss of significant digits or loss of precision | Possible loss of significant digits or loss of precision | Possible loss of significant digits or loss of precision | Possible loss of significant digits or loss of precision | | | See JavaDoc for the java.lang.Double method toString() | Compile time error | | Compile time error | Compile time error |
| **Double** | 0 -> False other values-> True | Possible loss of significant digits or loss of precision | Possible loss of significant digits or loss of precision | Possible loss of significant digits or loss of precision | Possible loss of significant digits or loss of precision | Possible loss of significant digits or loss of precision | | See JavaDoc for the java.lang.Double method toString() | Compile time error | | Compile time error | Compile time error |
| **String** | "False" -> False "True" -> True Other values -> | Numerical values, with possible loss of significant digits or | Numerical values, with possible loss of significant digits or | Numerical values, with possible loss of significant digits or | Numerical values, with possible loss of significant digits or | Numerical values, with possible loss of significant digits or | Numerical values, with possible loss of significant digits or | | Compile time error | | Compile time error | Compile time error |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Runtime error | loss of precision. Other values -> Runtime error | loss of precision. Other values -> Runtime error | loss of precision. Other values -> Runtime error | loss of precision. Other values -> Runtime error | loss of precision. Other values -> Runtime error | loss of precision. Other values -> Runtime error | | | | | |
| **Date** | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | | | Compile time error | Compile time error |
| **Variant** | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. | | Depending on the type of the actual value. Runtime error if not possible. | Depending on the type of the actual value. Runtime error if not possible. |
| **Array** | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | | Element types must be identical. Dimensions must match, unless target is not dimensioned. | Compile time error |
| **Object** | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | Compile time error | | Compile time error | Assignment compatible with base object or interface object implemented by the object. Otherwise |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | compiler time error. |

### Wider and Narrower Types

For numeric and types a wider type has a greater range of values it can represent and a narrower type a smaller range of values. Conversion of narrow to wider types and vice versa may cause a loss of precision. When Single and Double values are converted to Long or Integer, the values are truncated toward zero. The Boolean type can be considered representing the values -1 and 0 for the purposes of type width. It therefore represents the narrowest 'numeric' type.

| **Narrow** | | | | | | **Wide** |
|---|---|---|---|---|---|---|
| Boolean | Byte | Short | Integer | Long | Single | Double |

For object types a base object is considered the wider type, while a derived object is considered a narrower type.

## Program Units (aka Source Files)

SimpleProgramUnit :=
{ AliasDeclaration }
{ Declaration }
PropertiesSection

A Simple program unit is a sequence of declarations. Alias declarations are limited to the beginning of the Simple program unit. All Simple program units end with a properties section as described in the appendix. The properties section is supposed to be maintained by tools and should not be edited directly.
There are three different kinds of program units which differ in the content of their properties sections:

- object program units which define objects and can be derived from base objects,
- interface program units which define an interface and do not contain any implementations,
- form program units which define a form for a user interface.

Program units must be defined in a package. A package is a namespace which can be used to disambiguate program units with the same name. Packages can be nested. The package name of a nested package is the names of the packages joined together starting with the outermost package name followed by increasingly nested package names, separated by . (a dot).

# Declarations

Declaration :=
VariableDeclaration | ConstantDeclaration | FunctionDeclaration | ProcedureDeclaration | PropertyDeclaration | EventDeclaration | EventHandlerDeclaration

There are no implicit declarations.

## Variable Declarations

VariableDeclaration :=
[ **Static** ] **Dim** Identifier **As** Type { **,** Identifier **As** Type }

Variables can be declared as object data members or local variables. They are initialized with the corresponding default value for their type.
Object data members can be declared as static in which case all object instances refer to the same data member entity. Non-static (or instance) data members are allocated for each object instance.

```
' Good Examples: Variable Declarations

Dim data1 As Integer  ' instance data member
Static Dim data2 As Integer, data3 As Integer  ' object data member

Sub Foo()
  Dim localData As Integer  ' local variable
End Sub
```

```
' Bad Examples: Variable Declarations

Sub Foo()
  Static Dim localData As Integer  ' compile time error: no 'static' local variables
End Sub
```

## Constant Declarations

ConstantDeclaration :=
**Const** Identifier **As** Type **=** ConstantExpression { **,** Identifier **As** Type **=** ConstantExpression }

Constants must be declared as object data members. Their value must be set at the point of declaration.

```
' Good Examples: Constant Declarations

Const ZERO As Integer = 0
Const ONE As Integer = ZERO + 1, TWO As Integer = ONE + 1
```

```
' Bad Examples: Constant Declarations

Dim Zero As Integer

Const ZERO As Integer = Zero        ' compiler time error: constant value expected

Sub Foo()
  Const LOCAL_ZERO As Integer = 0   ' compile time error: no local constants
End Sub
```

## Functions and Procedure Declarations

FunctionDeclaration :=
[ **Static** ] **Function** Identifier **(** [ FormalArguments ] **) As** Type
Statements
**End Function**

ProcedureDeclaration :=
[ **Static** ] **Sub** Identifier **(** [ FormalArguments ] **)**
Statements
**End Sub**

FormalArguments := FormalArgument { , FormalArgument }
FormalArgument := [ **ByRef** | **ByVal** ] Identifier **As** Type

Functions and procedures are object members. They cannot be nested. The difference between a function and a procedure is that the function has a result value while the procedure does not. Functions implicitly declare a local variable of the same name as the function of the result type of the function. The value of that variable at the point of exit for the function is the result value of the function.

Unless otherwise declared, parameters are passed by value. They can optionally be annotated by the **ByVal** keyword. If they are instead annotated by the **ByRef** keyword, they will be passed by reference. In that case when an l-value is passed as the reference parameter, all changes to the actual argument in the callee will be reflected in the value of the l-value. If an r-rvalue is passed as a reference parameter then it will be treated as if passed by value.
Note that a read-only property passed as a reference parameter will cause a runtime error rather than a compile time error.

```
' Examples of function and procedure declarations

Sub ParenthesizedExpressionExamples()
  Dim par1 As Integer, par2 As Integer, par3 As Integer

  par1 = 1
  par2 = 2
  par3 = 3
  TestProcedure(par1, par2, par3, TestFunction())
  ' par1 still 1, par2 still 2, but par3 now 4, 4th actual argument wasn't an lvalue - so no notable
changes there
End Sub

Sub TestProcedure(par1 As Integer, ByVal par2 As Integer, ByRef par3 As Integer, ByRef par4 As Integer)
  par1 = 2
  par2 = 3
  par3 = 4
  par4 = 5
End Sub

Function TestFunction() As Integer
```

```
  TestFunction = 4     ' will return 4
End Function
```

## Property Declarations

PropertyDeclaration :=
**Property** Identifier **As** Type
[ **Get**
Statements
**End Get** ]
[ **Set**
Statements
**End Set** ]
**End Property**

Properties are object members. A property declaration may define a getter function and/or a setter function. If there is no setter function defined, the property is a read-only property and must not be referred to on the left hand side of an assignment expression. If there is no getter function defined, the property is a write-only property and must only be referred to on the right hand side of an assignment expression.
The getter function implicitly declares a local variable of the same name as the property. The value of that variable at the point of exit of the getter function is the value of the property. The setter function also implicitly declares a local variable of the name as the property. Upon entry into the setter method, this variable contains the value that was assigned to the property in an assignment expression. The getter and setter functions in a property declaration must not declare any formal arguments.

```
' Examples of property declaration

Dim currentStatus As String   ' Backing storage for Status property

Property Status As String
  Get
    Status = currentStatus    ' Returns the status
  End Get
  Set
    currentStatus = Status    ' Stores status into backing
    StatusLabel.Text = Status ' Updates StatusLabel
```

```
  End Set
End Property

Sub SomeProcedure()
  Status = "Great!"   ' Sets currentStatus to "Great!" and also shows "Great!" as the text of the
StatusLabel
End Sub
```

## Event Declarations

EventDeclaration :=
**Event** Identifier **(** [ FormalArguments ] **)**
**End Event**

Events are object members. An event declaration defines an event that can be raised with the RaiseEvent statement.

```
' Example for declaring an event

Event TestEvent(str As String)
End Event
```

## Event Handler Declarations

EventHandlerDeclaration :=
**Event** Identifier **.** Identifier **(** [ FormalArguments ] **)**
[ Statements ]
**End Event**

Event handlers are object members. An event handler is similar to a procedure, but cannot be invoked directly. Instead event handlers are invoked by the RaiseEvent statement from within the object that defines the event. An event handler is defined as a pair of an instance data member (first identifier after the **Event** keyword) and an event (second identifier). The instance data member must be declared within the same object as the event handler. The type of the instance data member must define the associated event.

```
' Example for declaring an event handler

Dim Obj As ObjectWithEvent    ' variable holding instance of an object defining TestEvent

Event Obj.TestEvent(str As String)
  StatusLabel = "Status: " & str
End Event
```

## Alias Declarations

AliasDeclaration :=
    **Alias** Identifier **=** QualifiedIdentifier

The Alias declaration defines a short name for a qualified name. Subsequent uses of the short name are equivalent to using the qualified name. The Alias declaration can only be used at the beginning of a source file before any variable, function, procedure, property or event declarations. It is a compile time error to redefine a short name.
All objects from the com.google.devtools.simple.runtime package or any of its subpackages are automatically aliased to their unquallified object name.

```
' Good Examples:
' You cannot only use Alias for shorter names but also to disambiguate names

Alias FirstObjectType = com.google.devtools.example1.ObjectType   ' object type defined in
com.google.devtools.example1
Alias SecondObjectType = com.google.devtools.example2.ObjectType  ' object type defined in
com.google.devtools.example2

Dim data1 As FirstObjectType
Dim data2 As SecondObjectType
```

```
' Bad Examples:
' Alias redefinitions are not allowed

Alias Form = com.google.devtools.example1.ObjectType   ' compile time error!
```

```
Alias ObjectType = com.google.devtools.example1.ObjectType
Alias ObjectType = com.google.devtools.example2.ObjectType    ' compile time error!
```

## Scope

Objects are defined within the scope of their package and are globally accessible. That means that an object with the same name may be redefined in a different package, but not within the same package.

Object members (data members, functions, procedure, property and event) declarations are globally accessible and may only be defined within object scope. Another object member (data member, function, procedure, property or event) of the same name must not be defined within the same object. In particular that means for functions, procedures, properties and events that there is no overloading. Functions (procedures, properties and events) may override declarations from a base object while data members must not be redefined in derived objects.

Function, procedure and event parameters are defined within the scope of the corresponding function, procedure or event. They must not be redefined within the parameter list.

Local variables are declared anywhere within the body of a function, procedure, property or event. Their scope starts at the point of their declaration spanning the remainder of the statement block containing their declaration and all nested statement blocks. Within the same statement block local variables must not be redeclared. A local variable declaration may shadow another local variable declaration of the same name in an outer statement block as well as data member declarations in the object that contains the enclosing event, function, procedure or property.

```
' Good example of scopes

Static Dim var1 As String
Static Dim var2 As Integer
Static Dim var3 As Object

Event ScopesTest.Load()
  var1 = "Not a number"
  var2 = 0
  var3 = Nothing
```

```
End Event

Static Sub TestProcedure(var1 As Integer)
  Dim var2 As String

  Do
    Dim var2 As Object
    Dim var3 As String

    var1 = com.google.devtools.simple.smoketest.scopes.ScopesTest.var2  ' integer to integer assignment
    var2 = ScopesTest.var3                                              ' object to object assignment
  While False
End Sub
```

# Expressions

## Operator Precedence

Operators are applied in the order of their precedence. Operators with high precedence are applied before operators with low precedence.

Operator Precedence (high to low):

- ^ (exponentiation)
- + - (identity and negation)
- * / (multiplication and division)
- \ (integer division)
- Mod (modulo)
- + - (addition and subtraction)
- & (string concatenation)
- << >> (bit shift)
- < <= > >= = <> Is IsNot Like TypeOf ... Is (comparison)
- Not (logical and bit operations - negation)
- And (logical and bit - conjunction)
- Or Xor (logical and bit operations - disjunction)

- = (assignment)

## Associativity

If an expression contains multiple operands and operators of the same precedence level then the operands and operators will be evaluated from left to right.

## Simple Expressions

PrimaryExpression := **Me** | Identifier | QualifiedIdentifier | Literal | ParenthesizedExpression | CallOrArrayExpression | NewExpression

### Me Expression

The **Me** keyword can be used in instance functions and procedures as well as properties and events to refer to the current instance object.

### Identifiers

Identifiers are defined in the lexical specification of the language are used to name symbols (such as variables, constants, namespaces, types, functions, procedures, properties and events).

### Qualified Identifiers

QualifiedIdentifier := [ PrimaryExpression **.** ] Identifier

There is two forms of qualified identifiers. The first is used to disambiguate an identifier in the case were it potentially refers to more than one symbol. In this case the primary expression denotes a package name. Packages name must always be absolute. In the other form of the qualified identifer the primary expression evaluates to an object instance and the identifier denotes a member of the object type of that object instance.

### Literals / Constant Expressions

ConstantExpression := Expression

A constant expression is any expression consisting of literals as defined in the lexical specification of the language in conjunction with any applicable operators.

### Call And Array Expressions

CallOrArrayExpression := QualifiedIdentifier **(** [ ActualArguments ] **)**
ActualArguments := Expression { **,** Expression }

A qualified identifier followed by open parenthesis is either a call or an array expression depending on the type the qualified identifier resolves to.
If the type of the qualified identifier resolves to an array type then it must be followed by a list of indices for the array access. The number of indices must match the number of dimensions of the array. The individual indices must be assignment compatible to the Integer type.
Otherwise the qualified identifier must resolve to a function or procedure name. It is then followed by a list of arguments for that function. The number of actual arguments must match the number of formal argument for the function. Each individual actual argument must be assignment compatible to its corresponding formal argument type.
The list of argument or indices is evaluated left to right.
The type of the call being made depends on the function or procedure it being resolved to. If the function or procedure is an object function or object procedure then it is invoked directly. If the function or procedure is an instance function or procedure then it will either be invoked via a virtual function table based on the actual type of the instance object at runtime. In the case where the instance object is an interface object the lookup will be made by name based on the actual type of the instance object at runtime.

```
' Examples of call and array expressions

Dim Array As Integer(2, 3)

Sub CallAndArrayExpressionExamples()
  Array(1, 2) = 3              ' Sets the value of the array element to 3
  Procedure(Array(1, 2), "4")  ' Reads the value of the array element (which is 3 now) and passes it to the
procedure
                               ' 2nd actual argument will be converted to an integer before passing it to
the procedure
End Sub
```

```
Sub Procedure(arg1 As Integer, arg2 As Integer)
End Sub
```

### Parenthesized Expressions

ParenthesizedExpression := **(** Expression **)**

Parenthesis can be used to override left associativity as well as operator precedence.

```
' Examples of parenthesized expressions

Sub ParenthesizedExpressionExamples()
  Dim result As Integer

  ' operator precedence
  result = 2 ^ 4 + 1   ' result is 17
  result = 2 ^ (4 + 1) ' result is 32

  ' left associativity
  result = 2 - 4 + 1   ' result is -1
  result = 2 - (4 + 1) ' result is -3
End Sub
```

### New Expressions

NewExpression := **New** QualifiedIdentifier [ **On** QualifiedIdentifier ] | **New** QualifiedIdentifier **(** Expression { **,** Expression } **)**

A New expression allocates and initializes an object or dynamically sized array instance.
After allocating an object instance and initializing any of its data members with their corresponding default values the New operator will raise the object instance initialization event. If the object being allocated is a component then the component name must be followed by the **On** keyword and the name of a component container into which the component should be placed. The component is placed into its container before its initialization event occurs.

When allocating a dynamically sized array, the sizes for each dimension will be passed as parameters to the **New** operator. After allocating an array instance any array elements will be initialized with their corresponding default values.

```
' Examples of new operator expressions

Sub NewExpressionExamples()
  Dim label As Label
  Dim obj As Object
  Dim array1 As Integer(), array2 As Integer(,)

  label = New Label On MainForm     ' places the label onto the main form
  obj = New SampleObject            ' allocates a new instance of SampleObject
  array1 = New Integer(5)           ' allocates a one-dimensional array with 5 Integer elements
  array2 = New Integer(2, 3)        ' allocates a two-dimensional array with a total of 6 Integer elements
End Sub
```

## Complex Expressions

Expression := LogicalOrBitOpExpression

Complex expressions consist of an operator which is applied to one or more operands.

### Exponentiation Operators

ExponentiationExpression := PrimaryExpression | PrimaryExpression **^** ExponentiationExpression

The binary **^** operator raises the left operand to the power of the right operand.
If necessary the operands will be converted to a Double type before applying the operation. The result of the operation is of type Double.
Floating point exponentiations follow the IEEE 754 arithmetic rules.
Disregarding possible conversion errors, the operation itself will cause no runtime errors, regardless of possible overflow, underflow, loss of precision or undefined values as a result of its application.

```
' Examples of exponentiation operator expressions

Sub ExponentiationExpressionExamples()
  Dim result As Integer

  result = 2 ^ 4 + 1    ' result is 17
  result = 2 ^ (4 + 1)  ' result is 32
End Sub
```

### Identity And Negation Operators

IdentityNegationExpression := [ + | - ] ExponentiationExpression

The unary **+** operator does not change the value of the operand. The binary **-** operator negates the value of its operand.
If necessary the operand will be converted to a numeric type (Byte, Short, Integer, Long, Single or Double) before applying the operation. The type of the result is the same as the type of the operand after any conversions.
Floating point negations follow the IEEE 754 arithmetic rules.
Disregarding possible conversion errors, the operation itself will cause no runtime errors, regardless of possible loss of precision as a result of its application.

```
' Examples of identity and negation operator expressions

Sub NegationExpressionExamples()
  Dim result As Integer

  result = 4        ' result is 4
  result = +result  ' result is 4
  result = -result  ' result is -4
End Sub
```

### Multiplication And Division Operators

MultiplicationExpresion := IdentityNegationExpression | IdentityNegationExpression MultiplicationOperator MultiplicationExpression
MultiplicationOperator := * | /

The binary * operator performs a multiplication of its operands.
If necessary the operands will be converted to a numeric type (Byte, Short, Integer, Long, Single or Double) before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. The type of the result is the same as the common type of the operands.
Floating point multiplications follow the IEEE 754 arithmetic rules.
Disregarding possible conversion errors, the operation itself will cause no runtime errors, regardless of possible overflow, underflow or loss of precision as result of its application.

The binary / operator performs the division of its left operand by its right operand.
If necessary the operands will be converted to a numeric type (Byte, Short, Integer, Long, Single or Double) before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. The result of the operation is of type Double.
Floating point divisions follow the IEEE 754 arithmetic rules.
Disregarding possible conversion errors, the operation can cause a runtime error if the right operand is zero, but no runtime errors for possible overflow, underflow or loss of precision as a result of its application.

```
' Examples of division and multiplication operator expressions

Sub MultiplicationExpressionExamples()
  Dim result As Integer

  result = 12 / 2 * 6   ' result is 36
  result = 12 / (2 * 6) ' result is 1
  result = 12.5 / 0     ' Will cause a runtime error
End Sub
```

### Integer Divison Operators

IntegerDivisionExpression := MultiplicationExpression | MultiplicationExpression \ IntegerDivisionExpression

The binary **** operator performs an integer division of its left operand by its right operand.
If necessary the operands will be converted to a numeric type (Byte, Short, Integer, Long, Single or Double) before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. The result of the operation is of type Integer.
Floating point divisions follow the IEEE 754 arithmetic rules.
Disregarding possible conversion errors, the operation can cause a runtime error if the right operand is zero, but no runtime errors for possible overflow, underflow or loss of precision as result of its application.

```
' Examples of integer division operator expressions

Sub IntegerDivisionExpressionExamples()
  Dim result As Integer

  result = 12.5 \ "3.1"    ' result is 4
  result = 12.5 \ 0        ' Will cause a runtime error
End Sub
```

### Modulo Operators

ModuloExpression := IntegerDivisionExpression | IntegerDivisionExpression **Mod** ModuloExpression

The binary **Mod** operator yields the remainder of the division of the left operand by the right operand, as defined by the formula (left / right) * right + (left Mod right) `= left`.
If necessary the operands will be converted to a numeric type (Byte, Short, Integer, Long, Single or Double) before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. The type of the result is the same as the common type of the operands.
Floating point modulo operations follow the IEEE 754 arithmetic rules.
Disregarding possible conversion errors, the operation can cause a runtime error if the right operand is zero, but no runtime errors for possible overflow, underflow or loss of precision as result of its application.

```
' Examples of modulo operator expressions

Sub ModuloExpressionExamples()
  Dim result As Integer
```

```
  result = 12 Mod 3    ' result is 0
  result = 11 Mod 3    ' result is 2
  result = -11 Mod 3   ' result is -2
  result = 12.5 Mod 0 ' Will cause a runtime error
End Sub
```

### Addition And Subtraction Operators

AdditionExpression := ModuloExpression | ModuloExpression AdditionOperator AdditionExpression
AdditionOperator := **+** | **-**

The binary **+** operator performs an addition of its operands. The binary **-** operator performs the subtraction of its right operand from its left operand.
If necessary the operands will be converted to a numeric type (Byte, Short, Integer, Long, Single or Double) before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. The type of the result is the same as the common type of the operands.
Floating point additions and subtractions follow the IEEE 754 arithmetic rules.
Disregarding possible conversion errors, the operation itself will cause no runtime errors, regardless of possible overflow, underflow or loss of precision as result of its application.

```
' Examples of addition and subtraction operator expressions

Sub AdditionExpressionExamples(arg As String)
  Dim result As Integer

  result = 2 + "5" - 1    ' result is now 6
  result = result + arg   ' this causes a runtime error if the contents of the arg string cannot be
converted to a number!
End Sub
```

### String Concatenation Operators

StringConcatenationExpression := AdditionExpression | AdditionExpression **&** StringConcatenationExpression

The binary & operator performs an concatenation of its two string operands. If necessary the operands will be converted to the string type before applying the operation. The result of the operation is of type String.
Disregarding possible conversion errors, the operation will cause no runtime errors.

```
' Examples of string concatenation operator expressions

Sub StringConcatentionExpressionExamples()
  Dim result As String

  result = "abc" & 2 + "5"    ' result is now "abc7"
End Sub
```

### Shift Operators

BitShiftExpression := StringConcatenationExpression | StringConcatenationExpression BitShiftOperator BitShiftExpression
BitShiftOperator := **<<** | **>>**

The binary **<<** operator performs a left shift of the left operand by the number of bits specified by the right operand. The binary **>>** operator performs a signed right shift of the left operand by the number of bits specified by the right operand.
If necessary the operands will be converted to an integer type (Byte, Short, Integer or Long) before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. The type of the result is the same as the common type of the operands.
Disregarding possible conversion errors, the operation will cause no runtime errors.

```
' Examples of bit shift operator expressions

Sub BitShiftExpressionExamples()
  Dim result As Integer

  result = &H12348080
  result = result << 16     ' result is now &H80800000
```

```
  result = result >> 16     ' result is now &HFFFF8080
End Sub
```

### Comparison, Is, IsNot, Like And TypeOf...Is Operators

ComparisonExpression := BitShiftExpression | BitShiftExpression ComparisonOperator ComparisonExpression | TypeOfExpression
TypeOfExpression := **TypeOf** PrimaryExpression **Is** Type
ComparisonOperator := OrderedComparisonOperator | **Is** | **IsNot** | **Like**
OrderedComparisonOperator := **<** | **<=** | **=** | **<>** | **>=** | **>**

The binary **<** operator indicates whether the left operand is numerically less then the right operand. The binary **<=** operator indicates whether the left operand is numerically less then or equal to the right operand. The binary **>=** operator indicates whether the left operand is numerically greater then the right operand. The binary **>** operator indicates whether the left operand is numerically greater then or equal to the right operand. If necessary the operands will be converted to an integer type (Byte, Short, Integer or Long) or the String type before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. If both operands are of type String then they will be compared lexicographically as defined by the compareTo() method of the *java.lang.String* class defined by Java™.
The binary **=** operator indicates whether the left operand is numerically equal to the right operand. The binary **<>** operator indicates whether the left operand is numerically different from then the right operand. If the operands have different types then the operand with the narrower type will also be converted to the wider type.
The comparison **Is** operator indicates whether the left operand is identical to the right operand. The comparison **IsNot** operator indicates whether the left operand is not identical to the right operand. The operands of the operation must be of Object or Array type.
The comparison **Like** operator indicates whether the left operand matches the regular expression given by the right operand. Regular expression syntax follows the definitions given by the *java.util.regex.Pattern* class defined by Java™. If necessary the operands will be converted to the String type before applying the operation.
The comparison **TypeOf...Is** operator indicates whether the left operand is assignment compatible to the type given by the right operand.
The result of the operation is of type Boolean.
Disregarding possible conversion errors, the operation will cause no runtime errors.

```
' Examples of comparison operator expressions

Sub ComparisonExpressionExamples()
```

```
  Dim result As Boolean

  result = "bar" < "foo"    ' result is True
  result = 2 < "one"        ' result is True
  result = "bar" = "foo"    ' result is False
  result = result <> True   ' result is True
End Sub

Sub IsExpressionExamples()
  Dim result As Boolean
  Dim obj1 As Object, obj2 As Object

  obj1 = SomeObject
  obj2 = SomeObject
  result = obj1 Is obj2     ' result is True

  obj2 = DifferentObject    ' Different object with same type and identical data member values
  result = obj1 Is obj2     ' result is False
End Sub

Sub LikeExpressionExamples()
 Dim result As Boolean

 result = "foof" Like "f.*f"  ' result is True
 result = "goof" Like "f.*f"  ' result is False
End Sub

Sub TypeOfExpressionExamples()
  Dim result As Boolean
  Dim obj As Object
  Dim var As Variant

  obj = Nothing
  var = 1
```

```
  result = TypeOf obj Is Integer    ' result is False
  result = TypeOf obj Is Integer()  ' result is False
  result = TypeOf obj Is Object     ' result is True
  result = TypeOf obj Is Foo        ' result is True

  obj = SomeObject
  result = TypeOf obj Is Foo        ' result is True if obj is of type Foo or of a derived type of Foo

  result = TypeOf var Is Integer    ' result is True
  result = TypeOf var Is String     ' result is True
  result = TypeOf var Is Object     ' result is False
End Sub
```

### Logical and Bit Operators

LogicalOrBitOpExpression := ComparisonExpression | ComparisonExpression LogicalOrBitOperator LogicalOrBitExpression
| **Not** LogicalOrBitOpExpression
LogicalOrBitOperator := **And** | **Or** | **Xor**

The binary **And** operator results in a bitwise AND of its operands. The binary **Or** operator results in a bitwise inclusive OR of its operands. The binary **Xor** operator results in a bitwise exclusive OR of its operands. If necessary the operands will be converted to an integer type (Byte, Short, Integer or Long) before applying the operation. If the operands have different types then the operand with the narrower type will also be converted to the wider type. The type of the result is the same as the common type of the operands.
The unary **Not** operator results in the bitwise complement of its operand. If necessary the operand will be converted to an integer type (Byte, Short, Integer or Long) before applying the operation.
Disregarding possible errors, the operation itself will cause no runtime errors as result of its application.
For operands of Boolean type, the operations behave like they would operate on a single bit operands.

```
' Examples of logical/bit operator expressions

Sub BitExpressionExamples()
  Dim result As Integer
```

```
  result = &H12348080
  result = result And &H0000FFFF  ' result is now &H00008080
  result = result Or  &H80800000  ' result is now &H80808080
  result = result Xor &H0000FFFF  ' result is now &H80807F7F
  result = Not result             ' result is now &H7F7F8080
End Sub

Sub LogicalExpressionExamples()
  Dim result As Boolean

  result = False
  result = result And True  ' result is now False
  result = result Or  True  ' result is now True
  result = result Xor True  ' result is now False
  result = Not result       ' result is now True
End Sub
```

### Assignment Operator

AssignmentExpression := LValue = Expression
LValue := QualifiedIdentifier | CallOrArrayExpression

The binary = operator assigns the right operand to the left operand. The left operand needs to be modifiable, meaning it needs to refer to a variable (local variable or data member), a property or an array element. If necessary the right operand will be converted to the same type as the left operand. If that conversion is possible then the operands are assignment compatible. Disregarding possible conversion errors, the operation itself will cause no runtime errors as result of its application.

```
' Good examples of assignment expressions

Sub GoodExpressionExample()
  Dim i As Integer
  Dim a As Integer(2)

  i = "1"     ' Assignment to variable
```

```
  a(i) = i     ' Assignment to array element
End Sub
```

```
' Bad examples of assignment expressions

Function One() As Integer
  One = 1
End Function

Sub BadExpressionExample()
  Dim i As Integer

  One() = 2   ' Compile time error: cannot assign to a function
  i + 1 = 3   ' Compile time error: cannot assign to expression
End Sub
```

## Statements

Statements := Statement | Statements
Statement := DoWhileStatement | DoUntilStatement | ExpressionStatement | ExitStatement | ForEachStatement | ForToStatement | IfStatement | OnErrorStatement | RaiseEventStatement | REMStatement | SelectStatement | WhileStatement | VariableDeclaration

The bodies of functions, event, properties and procedures consist of a sequence of statements and variable declarations.

### Do...While Statements

DoWhileStatement :=
**Do**
Statements
**While** Expression

The Do...While statement executes a block of statements, its body, repeatedly as long as the associated expression evaluates to **True**. The body is executed at least once, because the associated expression is evaluated after the body is executed. The execution of the Do...While statement can be aborted by executing an Exit statement inside its body.

```
' Example of a Do...While statement

Sub DoWhileExample()
  Dim counter As Integer
  ' counter is 0
  Do
    counter = counter + 1
  While counter < 20
  ' counter is 20
End Sub
```

## Do...Until Statements

DoUntilStatement :=
**Do**
Statements
**Until** Expression

The Do...Until statement executes a block of statements, its body, repeatedly as long as the associated expression evaluates to **False**. The body is executed at least once, because the associated expression is evaluated after the body is executed. The execution of the Do...Until statement can be aborted by executing an Exit statement inside its body.

```
' Example of a Do...Until statement

Sub DoUntilExample()
  Dim counter As Integer
  ' counter is 0
  Do
    counter = counter + 1
  Until counter >= 20
```

```
  ' counter is 20
End Sub
```

## Expression Statements

ExpressionStatement := AssignmentExpression | CallOrArrayExpression

An expression statement must either be an assignment expression or a call expression.

```
' Good examples of expression statements

Function One() As Integer
  One = 1
End Function

Sub GoodExpressionExample()
  Dim i As Integer
  Dim a As Integer(2)

  i = "1"       ' Assignment to variable
  a(i) = One()  ' Assignment to array element
  One()         ' Call of function - OK even though result unused
End Sub
```

```
' Bad examples of expression statements

Sub BadExpressionExample()
  Dim a As Integer(2)

  a(i)  ' Compile time error: neither an assignment expression nor a call expression - just an array access
End Sub
```

## Exit Statements

ExitStatement :=
**Exit** [ **Do** | **Event** | **For** | **Function** | **Property** | **Sub** | **While** ]

The Exit statement aborts the execution of its parent loop statement or function, procedure, property or event.
If no additional keyword follows the **Exit** keyword, then the compiler will look for the closest surrounding loop statement. If one is found, then that loop will be aborted upon execution of the Exit statement. If no loop is found, the enclosing function, procedure, property or event will be exited.
If a **Do**, **For** or **While** keyword follows the **Exit** keyword, then the compiler will look for the closest surrounding and matching loop statement. If one is found, then that loop will be aborted upon execution of the Exit statement. If no loop is found, a compile time error is reported.
If a **Event**, **Function**, **Property** or **Sub** keyword follows the **Exit** keyword, then the enclosing function, procedure, property or event will be exited. It is a compile time error if the keyword after the **Exit** keyword does not match its enclosing entity.

```
' Example of Exit statements

Function ExitFromDoWhileExample(exitFromFunction As Boolean) As Boolean
  Do
    If exitFromFunction Then
      ExitFromDoWhileExample = False
      Exit Function
    Else
      ExitFromDoWhileExample = True
      Exit ' Will exit the loop, but not the function
    End If
  While True
End Function
```

## For Each Statements

ForEachStatement :=
**For Each** <identifier> **In** <expression>
Statements
**Next** [ <identifier> ]

The For Each statement executes a block of statements, its body, once for each element in the array or collection given by the expression following the **In** keyword. Each element of the array or collection will be assigned to the loop variable associated with the identifier following the **Each** keyword. Note that the For Each statement does not implicitly define the loop variable. The loop variable must be of assignment compatible with the elements in the array or collection. The execution of the For Each statement can be aborted by executing an Exit statement inside its body. The end of the loop is marked by the **Next** keyword, optionally followed by the loop variable identifier.

```
' Example of For Each statements

Dim c As Collection
Dim a As Integer(2)

Event ForEachExample.Initialize()
  ' Initialize collection
  c = New Collection
  c.Add("one")
  c.Add(2)
  c.Add(Me)
End Event

' Example of a For Each statement for a collection
Function ForEachCollectionExample() As Integer
  Dim v As Variant
  For Each v In GetCollection()
    ForEachCollectionExample = ForEachCollectionExample + 1
  Next v
End Function

' Example of a For Each statement for an array
Function ForEachArrayExample() As Integer
  Dim i As Integer
  For Each i In GetArray()
    ForEachArrayExample = ForEachArrayExample + 1
  Next i
End Function

Function GetCollection() As Collection
  GetCollection = c
```

```
End Function

Function GetArray() As Integer()
  GetArray = a
End Function
```

## For...To Statements

ForToStatement :=
**For** <identifier> **=** Expression **To** Expression [ **Step** Expression ]
Statements
**Next** [ <identifier> ]

The For...To statement executes a block of statements, its body, until the loop variable associated with the identifier following the **For** keyword equals or exceeds the value of the expression following the **To** keyword (the end expression). At the beginning of the loop execution the loop variable will be initialized with the value of the expression between the **=** operator and the **To** keyword (initialization expression). After each execution of the statements of the loop body, the value of the loop variable will be incremented by 1 unless the For...To statement also contains the **Step** keyword, in which case the loop variable will be incremented by the value of the expression following the **Step** keyword (step expression). For a positively signed step expression value, the loop terminates if the loop variable is greater or equal to the value of the end expression value; for a negatively signed step expression value the loop terminates if the loop variable is less or equal to the end expression value. Both, the end and the step expression, are evaluated exactly once (and not repeatedly after each iteration). Note that the For...To statement does not implicitly define the loop variable. The loop variable must be of assignment compatible with the values in the initialization, end and step expressions. The execution of the For...To statement can be aborted by executing an Exit statement inside its body. The end of the loop is marked by the **Next** keyword, optionally followed by the loop variable identifier.

```
' Example of For...To statements

Function ForIntegerExample(start As Integer, stop As Integer) As Integer
  Dim i As Integer
  For i = start To stop
    ForIntegerExample = ForIntegerExample + i
  Next i
End Function

Function ForWithStepDoubleExample(start As Double, stop As Double, step As Double) As Double
```

```
  Dim d As Double
  For d = start To stop Step step
    ForWithStepDoubleExample = ForWithStepDoubleExample + d
  Next  ' No loop variable named after Next - that's OK too
End Function
```

## If Statements

IfStatement :=
    **If** Expression **Then** Statement [ **Else** Statement ]
|
    **If** Expression **Then**
      Statements
    { **ElseIf** Expression **Then**
      Statements }
    [ **Else**
      Statements ]
    **End If**

The If statement allows conditional execution of statements. If the value of the expression after the **If** keyword evaluates to true (after implicit conversion if necessary) then the statements after the Then keyword will be executed and the If statement terminates. Otherwise, if the expression evaluates to false and there are optional ElseIf then the expressions after the **ElseIf** keyword are evaluated until one is found that evaluates to true and the statements after the corresponding Then keyword will be executed and the If statement terminates. Finally, should none of the expressions following the If or ElseIf keyword evaluate to true, and there is an optional **Else** keyword, then the statements after the **Else** keyword will be executed. Otherwise none of the statements within the If statement will be executed.

```
' Example for the various forms of If statements

' The following constants are used as return values for the actual test functions. They indicate
' which statement block of the statement was actually executed.
Const NONE As String = ""
Const THEN As String = "Then"
Const ELSETHEN As String = "ElseThen"
Const ELSE As String = "Else"

' Example of a single line If statement without Else
```

```
Function SingleLineIfThenExample(testValue As Boolean) As String
  If testValue Then SingleLineIfThenExample = THEN
End Function

' Example of a single line If statement with Else
Function SingleLineIfThenElseExample(testValue As Boolean) As String
  If testValue Then SingleLineIfThenElseExample = THEN Else SingleLineIfThenElseExample = ELSE
End Function

' Example of a multi line If statement without Else
Function IfThenExample(testValueForThen As Boolean) As String
  If testValueForThen Then
    IfThenExample = THEN
  End If
End Function

' Example of a multi line If statement with Else
Function IfThenElseExample(testValueForThen As Boolean) As String
  If testValueForThen Then
    IfThenElseExample = THEN
  Else
    IfThenElseExample = ELSE
  End If
End Function

' Example of a multi line If statement with ElseIf and without Else
Function IfThenElseIfExample(testValueForIf As Boolean, testValueForElseIf As Boolean) As String
  If testValueForIf Then
    IfThenElseIfExample = THEN
  ElseIf testValueForElseIf Then
    IfThenElseIfExample = ELSETHEN
  End If
End Function

' Example of a multi line If statement with ElseIf and Else
Function IfThenElseIfElseExample(testValueForIf As Boolean, testValueForElseIf As Boolean) As String
  If testValueForIf Then
    IfThenElseIfElseExample = THEN
```

```
  ElseIf testValueForElseIf Then
    IfThenElseIfElseExample = ELSETHEN
  Else
    IfThenElseIfElseExample = ELSE
  End If
End Function
```

## On Error Statements

OnErrorStatement :=
On Error
{ Case Types
Statements }
[ Case Else
Statements ]
End Error

Types := Type { , Type }

The OnError statements allows handling of runtime errors. It must be the last statement of a function, procedure, event handler or property getter or setter. This implicitly means that there can be only one OnError statement per function. If the other statements of the function do not cause a runtime error, the OnError statement will not be executed.
In case of a runtime exeption, execution will continue at the OnError statement of the function in which the runtime error originated. If that function does not define an OnError statement then the runtime system will look for an OnError statement in the calling function. This is repeated until either an OnError statement is found or until the buttom of the call stack is reached, in which case the application will terminate with the runtime error. If an OnError statement is found, but it does not provide a handler for the runtime error, then the runtime error is passed through to the calling function.
An OnError statement defines an error handler in the statement block following a Case statement declaring a list of runtime error types to be handled. The special CaseElse statement will handle any runtime error.
It is a compile time error to define multiple error handlers for the same runtime error within the same OnError statement.

```
' Example of an OnError statement
' Function will return True if reading from the given array results in a runtime error

Function OnErrorExample(array As Integer(), index As Integer) As Boolean
  Dim value As Integer
```

```
  value = array(index)

  On Error
    Case AssertionFailure, UninitializedInstanceError
      ' Cannot happen, but just to demonstrate the syntax of the OnError statement

    Case ArrayIndexOutOfBoundsError
      ' Make function result indicate array out of bound
      OnErrorExample = True

    Case Else
      ' This can't happen either, but just to demonstrate the syntax of the OnError statement
  End Error
End Function
```

## RaiseEvent Statements

RaiseEventStatement :=
**RaiseEvent** Identifier **(** [ ActualArguments ] **)**

The RaiseEvent statement will invoke all event handlers with the same name as the identifier following the **RaiseEvent** keyword for the same implicit object instance of the instance function, property or procedure containing the RaiseEvent statement. The RaiseEventStatement can only be used in instance functions, properties and procedures of the object that defines the corresponding event.

```
' Example for raising a custom event

' Defines a custom event with an integer parameter
Event TestEvent(x As Integer)
End Event

' Raises the custom event passing the procedure parameter
Sub RaiseEventExample(x As Integer)
  RaiseEvent TestEvent(x)
End Sub
```

## Select Statements

SelectStatement :=
**Select** Expression
{ **Case** CaseExpressions
Statements }
[ **Case Else**
Statements ]
**End Select**

CaseExpressions := CaseExpression { **,** CaseExpression }

CaseExpression := Expression | **Is** OrderedComparisonOperator Expression | Expression **To** Expression

The Select statement allows conditional execution of statements. Depending on the value of the expression following the **Select** keyword (selector expression), the statements immediately following the first matching Case expression up to the next Case statement will be executed. After that, execution of the Select statement ends. This implies that none of the remaining case expressions will be evaluated.
If none of the Case expressions matches then none of the statements in the body of the Select statement will to be executed.
The types of the selector expression and the case expressions must be compatible.
Case expressions can be one or more, comma separated expressions. There are two special forms of expressions permitted. The first is a range expression which consists of two expressions, the first being a lower and the second being an upper bound, separated by the **To** keyword. Any selector expression value between the two bounds will match this case expression. The second form is a comparison expression, starting with the Is keyword and followed by a comparison operator and an expression. If the result of the comparison of the selector expression value and the comparison expression is true then the case expression matches.
Finally if the **Case** keyword is followed by the **Else** keyword then the statements following the Case statement will be executed regardless of the value of the selector expression. Note that this special form of the Case statement must be the last Case statement of the Select statement.

```
' Example of a Select statement: converting strings into numbers
Dim SpecialNumber As String

Function StringsToNumbers(number As String) As Integer
  Select number
    Case "Zero"
      StringsToNumbers = 0
```

```
    Case "One", SpecialNumber
      StringsToNumbers = 1
    Case Else
      StringsToNumbers = 2
  End Select
End Function

' Example of a Select statement: converting numbers into strings
Function NumbersToStrings(number As Integer) As String
  Select number
    Case Is < 0
      NumbersToStrings = "Negative number"
    Case 0
      NumbersToStrings = "Zero"
    Case 1
      NumbersToStrings = "One"
    Case 2 To 1000
      NumbersToStrings = "Between 2 and 1000"
    Case Else
      NumbersToStrings = "Big number"
  End Select
End Function
```

## While Statements

WhileStatement :=
**While** Expression
Statements
**End While**

The While statement executes a block of statements, its body, repeatedly as long as the associated expression evaluates to **True**. The execution of the While statement can be aborted by executing an Exit statement inside its body.

```
' Example of a While statement
```

```
Sub WhileExample()
  Dim counter As Integer
  ' counter is 0
  While counter < 20
    counter = counter + 1
  End While
  ' counter is 20
End Sub
```

# Appendix

## Inside Scoop on Simple Source Files - Properties Section

If you are just interested in using the Simple language then you can skip this section!

Every source file ends with a properties section. Form source files define the components used by the form and the properties of those components. Object source files define the base object (if there is one) and any implemented interfaces. Interface source files simply indicate that they are interfaces.

An editor supporting Simple should hide the properties sections of the source files. For form source files it should provide a visual designer that allows adding and removing components of the form. It should also provide a properties editor for the components. For object source files it should provide a properties editor to set a base object and any implemented interfaces. No other special support for interface source file is required.

### Properties Section Keywords

PropertiesSectionKeywords := **$As** | **$Define** | **$End** | **$Form** | **$Interface** | **$Object** | **$Properties** | **$Source**

### Form Source Files

FormSourcePropertiesSection :=

```
        $Properties
          $Source $Form
          $Define Identifier $As Form
            { ComponentPropertyOrNestedComponent }
          $End Define
        $End $Properties

    ComponentPropertyOrNestedComponent :=
        Identifier = Literal
        |
        $Define Identifier $As QualifiedIdentifier
          { ComponentPropertyOrNestedComponent }
        $End $Define
```

Only those properties whose value is different from their default value (as defined by the component implementation) need to be saved. For more information see the Component Writer's Guide.

```
' Example of a form source file's property section

$Properties
  $Source $Form
  $Define Main $As Form
    Title = "Hello World!"
    $Define ByeButton $As Button
      Text = "Bye..."
    $End Define
  $End $Define
$End $Properties
```

**Object Source Files**

```
    FormSourcePropertiesSection :=
        $Properties
          $Source $Object
          [ BaseObject = QualifiedIdentifier ]
```

{ **ImplementsInterface =** QualifiedIndentifier }
**$End $Properties**

```
' Example of an object source file's property section

$Properties
  $Source $Object
  BaseObject = com.yourdomain.Foo
  ImplementsInterface = com.somedomain.Interface1
  ImplementsInterface = com.otherdomain.Interface2
$End $Properties
```

### Interface Source Files

InterfaceSourcePropertiesSection :=
**$Properties**
**$Source $Interface**
**$End $Properties**

```
' Example of an interface source file's property section

$Properties
  $Source $Interface
$End $Properties
```