[MS-VBAL]:

VBA Language Specification

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation ("this documentation") for protocols, file formats, data portability, computer languages, and standards support. Additionally, overview documents cover inter-protocol relationships and interactions.
- **Copyrights**. This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you can make copies of it in order to develop implementations of the technologies that are described in this documentation and can distribute portions of it in your implementations that use these technologies or in your documentation as necessary to properly document the implementation. You can also distribute in your implementation, with or without modification, any schemas, IDLs, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications documentation.
- **No Trade Secrets**. Microsoft does not claim any trade secret rights in this documentation.
- **Patents**. Microsoft has patents that might cover your implementations of the technologies described in the Open Specifications documentation. Neither this notice nor Microsoft's delivery of this documentation grants any licenses under those patents or any other Microsoft patents. However, a given Open Specifications document might be covered by the Microsoft Open Specifications Promise or the Microsoft Community Promise. If you would prefer a written license, or if the technologies described in this documentation are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **License Programs**. To see all of the protocols in scope under a specific license program and the associated patents, visit the Patent Map.
- **Trademarks**. The names of companies and products contained in this documentation might be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.
- **Fictitious Names**. The example companies, organizations, products, domain names, email addresses, logos, people, places, and events that are depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than as specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications documentation does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments, you are free to take advantage of them. Certain Open Specifications documents are intended for use in conjunction with publicly available standards specifications and network programming art and, as such, assume that the reader either is familiar with the aforementioned material or has immediate access to it.

Support. For questions and support, please contact dochelp@microsoft.com.

Revision Summary

Date	Revision History	Revision Class	Comments
6/30/2008	0.9	Major	First release. Additional indexing and cross referencing as well as minor editorial and technical edits anticipated prior to 1.0 release.
6/30/2009	0.95	Major	Updated to include preliminary information on the VBA language from the pre-release version of VBA 7.
3/15/2010	1.0	Major	Updated to include information on the VBA language as of VBA 7.
3/15/2012	1.01	Major	Updated to include information on the VBA language as of VBA 7.1, as shipped in the Office 15 Technical Preview.
4/30/2014	1.02	Editorial	Revised and edited technical content.
12/15/2016	1.02	None	No changes to the meaning, language, or formatting of the technical content.
6/18/2019	1.3	Minor	Clarified the meaning of the technical content.
9/24/2019	1.4	Minor	Clarified the meaning of the technical content.

Table of Contents

1 Introduction

This specification defines the Visual Basic for Applications (VBA) Language, an implementation-independent and operating system-independent programming language that is intended to be imbedded as a macro language within host applications. This specification includes all features and behaviors of the language that exist and behave identically in all conforming implementations. Such features include the intrinsic functions that exist in conforming implementations.

1.1 Glossary

This document uses the following terms:

code page: An ordered set of characters of a specific script in which a numerical index (code-point value) is associated with each character. Code pages are a means of providing support for character sets and keyboard layouts used in different countries. Devices such as the display and keyboard can be configured to use a specific code page and to switch from one code page (such as the United States) to another (such as Portugal) at the user's request.

Unicode: A character encoding standard developed by the Unicode Consortium that represents almost all of the written languages of the world. The **Unicode** standard [UNICODE5.0.0/2007] provides three forms (UTF-8, UTF-16, and UTF-32) and seven schemes (UTF-8, UTF-16, UTF-16 BE, UTF-16 LE, UTF-32, UTF-32 LE, and UTF-32 BE).

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as defined in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information.

[IEEE754] IEEE, "IEEE Standard for Binary Floating-Point Arithmetic", IEEE 754-1985, October 1985, http://ieeexplore.ieee.org/servlet/opac?punumber=2355

[MS-DTYP] Microsoft Corporation, "Windows Data Types".

[MS-OAUT] Microsoft Corporation, "OLE Automation Protocol".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[RFC4234] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005, http://www.rfc-editor.org/rfc/rfc4234.txt

1.2.2 Informative References

[CODEPG] Microsoft Corporation, "Code Pages", https://docs.microsoft.com/en-us/globalization/encoding/code-pages

[UNICODE-BESTFIT] The Unicode Consortium, "WindowsBestFit", 2006, http://www.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WindowsBestFit/

[UNICODE-README] The Unicode Consortium, "Readme.txt", 2006, http://unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WindowsBestFit/readme.txt

1.3 VBA Language Specification Overview

VBA is a computer programming language that is intended to be used in conjunction with a host software application such as a word processor. In such a situation, the end-user of such a host application uses the VBA language to write programs that can access and control the host application's data and functionality.

This document is an implementation-independent specification of the VBA language that enables the creation of independent implementations. It enables the creation of source code compatible implementations of the language by defining the required characteristics and behaviors of the source language that is supported by all conforming implementations. It enables a programmer to write portable VBA programs by defining the exact set of implementation independent characteristics and behaviors of the language that a program can use if it is intended to run on multiple implementations.

The scope of the VBA Language Specification is the implementation independent, operating system independent core programming language that is supported by all conforming VBA implementations. It includes all features and behaviors of the language that exist and behave identically in all conforming implementations. Such features include the intrinsic functions that exist in conforming implementations.

This specification defines the syntax, static semantics, and runtime semantics of the VBA language. *Syntax* defines the source code representation of VBA programs that is recognized by a VBA implementation. *Static semantics* define non-syntactic program validity requirements that cannot be expressed using the grammar. *Runtime semantics* define the computational behavior of VBA programs that conform to the specified syntax and static semantics rules. The runtime semantics describes *what* it means to execute a VBA program but not *how* a VBA implementation might accomplish this.

The VBA Language Specification does not define how a VBA implementation would actually achieve the requirements of the specification nor does it describe the specific design of any VBA Language Implementation.

The language defined by this specification is that language implemented by Microsoft Visual Basic for Applications 7.0 (VBA 7.0), as shipped in Microsoft Office 2010 suites; and VBA version 7.1, as shipped in Microsoft Office 2013. It includes features that provide source code backward-compatibility for VBA programs written for prior Microsoft versions of VBA.

1.4 Specification Conventions

Lexical and syntactic constructs of the language are described by a grammar using ABNF as defined in [RFC4234] with additional conventions as defined in the introductions to sections 3 and 5 of this document. Within the prose text of this specification the names of ABNF rules are distinguished by enclosing them angle brackets, for example <for-statement>.

Static semantics rules are expressed as prose descriptions, tables, and pseudo code algorithms that reference grammar rules. Runtime semantics are expressed in prose using implementation independent abstract computational concepts.

This specification defines a large number of terms that have specialized meaning within the context of this specification. Such terms are generally italicized when used within this document. The first use of

each such term within a section of this document references the document section that defines the term.

Within this specification the phrase "implementation-defined" means that the contextually apparent detail of the syntax or semantics of a feature of the language is intentionally left unspecified and can vary among implementation of the language. However, the implementation of the unspecified details SHOULD be repeatedly consistent and the implementation SHOULD document its specific behavior order to preserve the utility of the language feature.

The phrase "implementation-specific" means that the contextually apparent detail of the syntax or semantics of a feature of the language is intentionally left unspecified and can vary among implementation of the language. However, the implementation of the unspecified details SHOULD be repeatedly consistent.

The phrase "undefined" means that the contextually apparent detail of the syntax or semantics of a feature of the language is intentionally left unspecified and can vary among implementation of the language. There is no requirement or expectation of consistent or repeatable behavior.

2 VBA Computational Environment

VBA is a programming language used to define computer programs that perform computations that occur within a specific computational environment called a *VBA Environment*. A *VBA Environment* is typically hosted and controlled by another computer application called the *host application*. The *host application* controls and invokes computational processes within its hosted *VBA Environment*. The *host application* can also make available within its hosted *VBA Environment* computational resource that enable VBA program code to access *host application* data and host computational processes. The remainder of this section defines the key computational concepts of the *VBA Environment*.

2.1 Data Values and Value Types

Within a *VBA Environment*, information is represented as data values. A *data value* is a single element from a specific finite domain of such elements. The *VBA Environment* defines a variety of *value types*. These value types collectively define the domain of VBA data values. Each value type has unique characteristics that are defined by this specification. Each data value within a *VBA Environment* is a domain member of one of these value types. Individual data values are immutable. This means that there are no defined mechanisms available within a *VBA Environment* that can cause a data value to change into another data value. Because data values are immutable, multiple copies of a specific data value can exist within a *VBA Environment* and all such copies are logically the same data value.

The value types of the *VBA Environment* are defined by the following table. The nominal representation is the representation that was used as the original design basis for the VBA value types.

Implementations can use these specific data type representations to meet the requirements of this specification.

Value Type Name	Domain Elements	Nominal Representation
Boolean	The distinguished values **True** and **False**	16-bit signed binary 2's complement integer whose value is either 0 (**False**) or -1 (**True**)
Byte	Mathematical integer in the range of 0 to 255	8-bit unsigned binary integer
Currency	Numbers with 4 fractional decimal digits in the range -922,337,203,685,477.5808 to +922,337,203,685,477.5807	64-bit signed binary two's complement integer implicitly scaled by 10^{-4}
Date	Ordinal fractional day between the first day of the year 100 and the last day of the year 9999.	8 byte IEEE 754-1985 [IEEE754] floating point value. The floating point value 0.0 represents the epoch date/time which is midnight of December 30, 1899. Other dates are represented as a number of days before (negative values) or after (positive value) the epoch. Fractional values represent fractional days.

Value Type Name	Domain Elements	Nominal Representation
Decimal	Scaled integer numbers whose maximum integer range is ±79,228,162,514,264,337,593,543,950,335. Number in this range MAY be scaled by powers of ten in the range 10^0 to 10^{-28}	A rational number represented in a 14 byte data structure including a sign bit and a 96-bit unsigned integer numerator. The denominator is an integral power of ten with an exponent in the range of 0 to 28 encoded in a single byte.
Double	All valid IEEE 754-1985 double-precision binary floating-point numbers including sized zeros, NaNs and infinities	64-bit hardware implementation of IEEE 7541985.
Integer	Integer numbers in the range of -32,768 to 32,767	16-bit binary two's complement integers
Long	Integer numbers in the range of -2,147,483,648 to 2,147,486,647	32-bit binary two's complement integers
LongLong	Integer numbers in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	64-bit binary two's complement integers
Object reference	Unique identifiers of *host application* or program created objects and a distinguished value corresponding to the reserved identifier **Nothing**	Machine memory addresses with the 0 address reserved to represent **Nothing**.
Single	All valid IEEE 754-1985 single-precision binary floating-point numbers including signed zeros, NaNs and infinities	32-bit hardware implementation of IEEE 7541985.
String	The zero length empty string and all possible character sequences using characters from the implementation dependent character set. There MAY be an implementation defined limit to the length of such sequences but the limit SHOULD be no smaller than $(2^{16} - 1)$ characters.	Sequences of 16-bit binary encoded **Unicode** code points.
Empty	A single distinguished value corresponding to the reserved identifier **Empty**	An implementation-specific bit pattern
Error	Standard error codes from 0 to 65535, as well as other implementation-defined error values. An implementation-defined error value can resolve to a standard error code from 0 to 65535 in a context where its value is required, such as **CInt**.	32-bit integer (Windows HRESULT)
Null	A single distinguished value corresponding to the reserved identifier **Null**	An implementation specific bit pattern

Value Type Name	Domain Elements	Nominal Representation
Missing	A single distinguished value corresponding that is used to indicated that no value was passed corresponding to an explicitly declared optional parameter.	An implementation specific bit pattern
An Array type	Multi-dimensional numerically indexed aggregations of data values with up to 60 dimensions. Empty aggregations with no dimensions are also included in the domain. Such aggregations can be homogeneous (all *elements (section 2.1.1)* of the aggregation have the same value type) or heterogeneous (the value types of elements are unrelated). Elements of each dimension are identified (indexed) via a continuous sequence of signed integers. The smallest index value of a dimension is the dimension's *lower bound* and the largest index value of a dimension is the dimension's *upper bound*. A lower bound and an upper bound might be equal.	A linear concatenation of the aggregated data values arranged in row major order possibly with implementation defined padding between individual data values.
A User-Defined Type (UDT)	Aggregations of named data values with possibly heterogeneous value types. Each UDT data value is associated with a specific named UDT declaration which serves as its value type.	A linear concatenation of the aggregated data values possibly with implementation defined padding between data values.

The VBA language also provides syntax for defining what appears to be an additional kind of data type known as an **Enum**. There is no **Enum**-specific value type. Instead, **Enum** members are represented as **Long** data values.

An implementation of VBA MAY include for other implementation-defined value types which can be retrieved as return values from procedures in referenced libraries. The semantics of such data values when used in statements and expressions within the *VBA Environment* are implementation-defined.

2.1.1 Aggregate Data Values

Data values (section 2.1) with a *value type (section 2.1)* of either a specific Array or a specific UDT name are *aggregate data values*. Note that object references are not aggregate data values. An aggregate data value consists of zero or more *elements* each corresponding to an individual data value within the aggregate data value. In some situations, an element is itself an aggregate data value with its own elements.

Each element of an aggregate data value is itself a data value and has a corresponding *value type*. The *value type* of an element is its *element type*. All elements of an Array data value have the same element type, while elements of an UDT data value can have differing value types.

2.2 Entities and Declared Types

An *entity* is a component of a *VBA Environment* that can be accessed by name or index, according to the resolution rules for simple name expressions, index expressions and member access expressions. Entities include projects, procedural modules, types (class modules, UDTs, Enums or built-in types), properties, functions, subroutines, events, variables, literals, constants and conditional constants.

For many kinds of entities, it is only valid to reference an entity that is *accessible* from the current context. Entities whose accessibility can vary have their accessibility levels defined in later sections specific to these entities.

Most entities have an associated a *declared type*. A declared type is a restriction on the possible *data values (section 2.1)* that a *variable (section 2.3)* can contain. Declared types are also used to restrict the possible data values that can be associated with other language entities. Generally declared types restricts the data value according to the data value's *value type (section 2.1)*.

The following table defines the VBA declared types. Every variable within a *VBA Environment* has one of these declared types and is limited to only containing data values that conform to the declared type's data value restrictions.

Declared Type	Data Value Restrictions
Boolean, Byte, Currency, Date, Double, Integer, Long, LongLong, Object, Single, or String	Only data values whose value type has the same name as the declared type. Note the following: ▪ Decimal is *not* a valid declared type. ▪ LongLong is a valid declared type only on VBA implementations that support 64-bit arithmetic.
Variant	No restrictions, generally any data value with any value type. However, in some contexts Variant declared types are explicitly limited to a subset of possible data values and value types.
String*n, where n is an integer between 1 and 65,535	Only data values whose value type is String and whose character length is exactly n.
Fixed-size array whose declared element type is one of Boolean, Byte, Currency, Date, Double, Integer, Long, LongLong, Object, Single, String, String*n, a specific class name, or the name of a UDT.	Only homogeneous array data values that conform to the following restrictions: ▪ The value type of every *element (section 2.1.1)* data value is the same as the variable's declared element type. If the variable's element declared type is a specific class name then every element of the data value MUST be either the object reference **Nothing** or a data value whose value type is object reference and which identifies either an object that is an *instance (section 2.5)* of the named element class or an object that *conforms (section 2.5)* to the *public interface (section 2.5)* of the named class. ▪ The number of dimensions of the data value is the same as the variable's number of dimensions. ▪ The *upper* and *lower bounds (section 2.1)* are the same for each dimension of the data value and the variable.

Declared Type	Data Value Restrictions
Fixed-size array whose declared element type is Variant	Only data values whose value type is Array and that conform to the following restrictions: ▪ The number of dimensions of the data value is the same as the variable's number of dimensions. ▪ The upper and lower bounds are the same for each dimension of the data value and the variable.
Resizable array whose declared element type is one Boolean, Byte, Currency, Date, Double, Integer, Long, LongLong, Object, Single, String, String*n, a specific class name, or the name of a UDT	Only homogeneous array data values where the value type of every element data value is the same as the variable's declared element type. If the variable's element declared type is a specific class name then every element of the data value MUST be either the object reference Nothing or a data value whose value type is object reference and which identifies either an object that is an instance of the named element class or an object that conforms to the public interface of the named class.
Resizable array whose declared element type is Variant	Only data values whose value type is Array.
Specific class name	Only the object reference data value Nothing and those data values whose value type is object reference and which identify either an object that is an instance of the named class or an object that conforms to the public interface of the named class.
Specific UDT name	Only data values whose value type is the specific named UDT.

As with value types, there is no **Enum**-specific declared type. Instead, declarations using an **Enum** type are considered to have a declared type of **Long**. Note that there are no extra data value restrictions on such **Enum** declarations, which might contain any **Long** data value, not just those present as **Enum** members within the specified **Enum** type.

An implementation-defined **LongPtr** type alias is also defined, mapping to the underlying declared type the implementation will use to hold pointer or handle values. 32-bit implementations SHOULD map **LongPtr** to **Long**, and 64-bit implementations SHOULD map **LongPtr** to **LongLong**, although implementations MAY map **LongPtr** to an implementation-defined pointer type. The **LongPtr** type alias is valid anywhere its underlying declared type is valid.

Every *declared type* except for array and UDT declared types are *scalar declared types*.

2.3 Variables

Within a *VBA Environment*, a *variable* is a mutable container of *data values (section 2.1)*. While individual data values are immutable and do not change while a program executes, the data value contained by a particular *variable* can be replaced many times during the program's execution.

Specific *variables* are defined either by the text of a VBA program, by the *host application*, or by this specification. The definition of a *variable* includes the specification of the *variable's declared type (section 2.2)*.

Variables have a well-defined lifecycle, they are created, become available for use by the program, and are then subsequently destroyed. The span from the time a *variable* is created to the time it is destroyed is called the *extent* of the *variable*. *Variables* that share a creation time and a destruction time are can be said to share a common *extent*. The *extent* of a *variable* depends upon how it was defined but the possible *extents* are defined by the following table.

Extent Name	Variable Definition Form	Variable Lifespan
Program Extent	Defined by the VBA specification or by the *host application.*	The entire existence of an active *VBA Environment*.
Module Extent	A Module Variable Declaration or a static local variable declaration within a procedure.	The span from the point that the containing *module* is incorporated into an active VBA project to the point when the *module* or *project* is explicitly or implicitly removed from its *VBA Environment*.
Procedure Extent	A procedure local *variable* or formal parameter declaration of a procedure.	The duration of a particular procedure invocation.
Object Extent	A *variable* declaration within a class module.	The lifespan of the containing object.
Aggregate Extent	A *dependent variable (section 2.3.1)* of an array or UDT variable.	The lifespan of the *variable* holding the containing *aggregate data value (section 2.1.1)*.

When a *variable* is created, it is initialized to a default value. The default value of a *variable* is determined by the *declared type* of the *variable* according to the following table.

Declared Type	Initial Data Value
Boolean	**False**
Byte, Currency, Double, Integer, Long, LongLong	0 value of the corresponding *value type (section 2.1)*
Double or Single	+0.0 value of the corresponding value type
Date	30 December 1899 00:00:00
String	The empty string
Variant	**Empty**
String*n, where n is an integer between 1 and 65,535	A string of length n consisting entirely of the implementation dependent representation of the null character corresponding to **Unicode** codepoint U+0000.
Fixed size array whose declared element type is one of Boolean, Byte, Currency, Data, Double, Object, Single, String, or String*n	The array data value whose number of dimensions and bounds are identical with the array's declared dimensions and bounds and whose every element is the default data value of the declared element type.

Declared Type	Initial Data Value
Fixed size array whose declared element type is Variant	The array value whose number of dimensions and bounds are identical with the array's declared dimensions and bounds and whose every element is the value **Empty**.
Resizable array whose declared element type is one of Boolean, Byte, Currency, Data, Double, Object, Single, String, or String*n	An array value with no dimensions.
Resizable array whose declared element type is Variant	An array value with no dimensions.
Object or a Specific class name	The value **Nothing**.
Specific UDT name	The UDT data value for the named UDT type whose every named element has the default data value from this table that is appropriate for that element's declared type.

Variables generally have a single *variable name* that is used to identify the *variable* within a VBA program. However, *variable names* have no computational significance. Some situations such as the use of a *variable* as a reference parameter to a procedure invocation can result in multiple names being associated with a single *variable*. Access to *variables* from within a VBA program element is determined by the visibility scopes of *variable names*. Typically, a *variable name*'s visibility is closely associated with the *variable*'s *extent* but *variable name* scopes themselves have no computational significances.

2.3.1 Aggregate Variables

A *variable (section 2.3)* that contain an *aggregate data value (section 2.1.1)* is an *aggregate variable*. An *aggregate variable* consists of *dependent variables* each one corresponding to an *element (section 2.1.1)* of its current *aggregate data value*. The *data value* contained by each *dependent variable* is the corresponding *element data value* of its containing *aggregate data value*. In some situations, a *dependent variable* itself holds an *aggregate data value* with its own *dependent variables*. *Dependent variables* do not have names; instead they are accessed using index expressions for arrays or member access expressions for UDTs.

When a new *data value* is assigned to a *dependent variable*, the *aggregate variable* holding this *dependent variable's* containing *aggregate data value* has its *data value* replaced with a new *aggregate data value* that is identical to its previous *data value* except that the *element data value* corresponding to the modified *dependent variable* is instead the *data value* being stored into the *dependent variable*. If this containing *aggregate data value* is itself contained in a *dependent variable* this process repeats until an *aggregate variable* that is not also a *dependent variable* is reached.

2.4 Procedures

A *procedure* is the unit of algorithmic functionality within a *VBA Environment*. Most *procedures* are defined using the VBA language, but the *VBA Environment* also contains standard *procedures* defined by this specification and can contain *procedures* provided in an implementation defined manner by the

host application or imported from externally defined libraries. A procedure is identified by a *procedure name* that is part of its declaration.

VBA also includes the concept of a *property*, a set of identically named *procedures* defined in the same *module (section 4.2)*. Elements of such a set of *procedures* can then be accessed by referencing the *property name* directly as if it was a *variable name (section 2.3)*. The specific *procedure* from the set that to be invoked is determined by the context in which the *property name* is referenced.

A *VBA Environment* is not restricted to executing a single program that starts with a call to a main procedure and then continues uninterrupted to its completion. Instead, VBA provides a reactive environment of variables, procedures, and objects. The *host application* initiates a computation by calling procedures within its hosted *VBA Environment*. Such a procedure, after possibly calling other procedures, eventually returns control to the *host application*. However, a *VBA Environment* retains its state (including the content of most variables and objects) after such a *VBA Environment* initiated call returns to the *host application*. The *host application* can subsequently call the same or other procedures within that *VBA Environment*. In addition to explicit *VBA Environment* initiated calls, VBA procedures can be called in response to *events (section 2.5)* associated with *host application*-provided objects.

2.5 Objects

Within the *VBA Environment*, an *object* is a set of related *variables (section 2.3)*, *procedures (section 2.4)* and events. Collectively, the variables, procedures and events that make up an object are called the object's *members*. The term *method* can be used with the same meaning as procedure member. Each object is identified by a unique identifier which is a *data value (section 2.1)* whose *value type (section 2.1)* is object reference. An object's members are accessed by invoking methods and evaluating member variables and properties using this object reference. Because a specific data value can simultaneously exist in many variables there can be many ways to access any particular object.

An object's *events* are attachment points to which specially named procedures can be dynamically associated. Such procedures are said to *handle* an object's events. Using the **RaiseEvent** statement of the VBA language, methods of an object can call the procedures handling a member event of the object without knowing which specific procedures are attached.

All variables and events that make up an object have the same *extent (section 2.3)* which begins when the containing object is explicitly or implicitly created and concludes when it is provably inaccessible from all procedures.

A *class* is a declarative description of a set of objects that all share the same procedures and have a similar set of variables and events. The members of such a set of objects are called *instances* of the class. A typical class can have multiple instances but VBA also allows the definition of classes that are restricted to having only one instance. All instances of a specific class share a common set of variable and event declarations that are provided by the class but each instance has its own unique set of variables and events corresponding to those declarations.

The access control options of VBA language declarations can limit which procedures within a *VBA Environment* are permitted to access each object member defined by a class. A member that is accessible to all procedures is called a *public member* and the set of all public procedure members and variable members of a class is called the *public interface* of the class. In addition to its own public interface the definition of a class can explicitly state that it implements the public interface of one or more other classes. A class or object that is explicitly defined to implement a public interface is said to *conform* to that interface. In this case the conforming class MUST include explicitly tagged definitions for all of the public procedure and variable members of all of the public interfaces that it implements.

When a variable is defined with the name of a class as its *declared type (section 2.2)* then that variable can only contain object references to instances of that specific named class or object references to objects that conform to the public interface of the named class.

2.5.1 Automatic Object Instantiation

A *variable (section 2.3)* that is declared with the name of a *class (section 2.5)* as its *declared type (section 2.2)* can be designated using the **New** *keyword (section 3.3.5.1)* to be an *automatic instantiation variable*. Each time the content of an automatic instantiation variable is accessed and the current data value of the variable is **Nothing**, a new *instance (section 2.5)* of the named class is created and stored in the variable and used as the accessed value.

Each *dependent variable (section 2.3.1)* of an array variable whose *element type (section 2.1.1)* is a named class and whose declaration includes the **New** *keyword* are automatic instantiation variables.

A class can also be defined such that the class name itself can be used as if it was an automatic instantiation variable. This provides a mechanism for accessing default instances of a class.

2.6 Projects

All VBA program code is part of a *project (section 4.1)*. A *VBA Environment* can contain one or more named projects. Projects are created and loaded into a *VBA Environment* using implementation defined mechanisms. In addition, a *VBA Environment* MAY include implementation mechanisms for modifying and/or removing projects.

2.7 Extended Environment

In addition to the *entities (section 2.2)* defined using VBA source code within VBA *projects (section 4.1)*, a *VBA Environment* can include entities that are defined within other sources and using other mechanisms. When accessed from VBA program code, such external environmental entities appear and behave as if they were environmental entities implemented using the VBA language.

2.7.1 The VBA Standard Library

The *VBA Standard Library (section 6)* is the set of *entities (section 2.2)* that MUST exist in all *VBA Environments*.

No explicit action is required to make these entities available for reference by VBA language code.

2.7.2 External Variables, Procedures, and Objects

In addition to *entities (section 2.2)* that are explicitly defined using VBA programming language, a *VBA Environment* can contain entities that have been defined using other programming languages. From the VBA language perspective such entities are consider to be defined by external libraries whose characteristics and nature is implementation defined.

2.7.3 Host Environment

A *host application*, using implementation-dependent mechanisms, can define additional *entities (section 2.2)* that are accessible within its hosted *VBA Environment*. Depending upon the VBA implementation and *host application*, such entities can be directly accessible similar to the *VBA Standard Library (section 2.7.1)* or can appear as external libraries or predefined VBA *projects (section 2.6)*.

The *host application* in conjunction with the VBA implementation is also responsible for providing the mapping of the VBA file I/O model to an application specific or platform file storage mechanism.

3 Lexical Rules for VBA Programs

VBA programs are defined using text files (or other equivalent units of text) called *modules (section 4.2)*. The role of modules in defining a VBA program is specified in section 4. This section describes the lexical rules used to interpret the text of modules.

The structure of a well-formed VBA module is defined by a set of inter-related grammars. Each grammar individually defines a distinct aspect of VBA modules. The grammars in the set are:

- The Physical Line Grammar
- The Logical Line Grammar
- The Lexical Token Grammar
- The Conditional Compilation Grammar
- The Syntactic Grammar

The first four of these grammars are defined in this section. The Syntactic Grammar is defined in section 5.

The grammars are expressed using ABNF [RFC4234]. Within these grammars numeric characters codes are to be interpreted as **Unicode** code points.

3.1 Character Encodings

The actual character set standard(s) used to externally encode the text of a VBA *module (section 4.2)* is implementation defined. Within this specification, the lexical structure of VBA modules are described as if VBA modules were encoded using **Unicode**. Specific characters are identified in this specification in terms of Unicode code points and character classes. The equivalence mapping between Unicode and an implementation's specific character encoding is implementation defined. Implementations using non-Unicode encoding MUST support at least equivalents to Unicode code points U+0009, U+000A, U+000D and U+0020 through U+007E. In addition, an equivalent to U+0000 MUST be supported within **String** data values as fixed-length strings are filled with this character when initialized.

3.2 Module Line Structure

The body of a VBA *module (section 4.2)* consists of a set of physical lines described by the *Physical Line Grammar*. The terminal symbols of this grammar are **Unicode** character code points.

3.2.1 Physical Line Grammar

```
module-body-physical-structure = *source-line [non-terminated-line]
source-line = *non-line-termination-character line-terminator
non-terminated-line = *non-line-termination-character
line-terminator = (%x000D  %x000A) / %x000D / %x000A / %x2028 / %x2029
non-line-termination-character = <any character other than %x000D / %x000A / %x2028 / %x2029>
```

An implementation MAY limit the number of characters allowed in a physical line. The meaning of a module that contains any physical lines that exceed such an implementation limit is undefined by this specification. If a <module-body-physical-structure> concludes with a <non-terminated-line> then an implementation MAY treat the module as if the <non-terminated-line> was immediately followed by a <line-terminator>.

For the purposes of interpretation as VBA program text, a *module body (section 4.2)* is viewed as a set of *logical lines* each of which can correspond to multiple physical lines. This structure is described by the *Logical Line Grammar*. The terminal symbols of this grammar are **Unicode** character codepoints.

3.2.2 Logical Line Grammar

```
module-body-logical-structure = *extended-line
extended-line = *(line-continuation / non-line-termination-character)  line-terminator
line-continuation = *WSC underscore *WSC line-terminator
WSC = (tab-character / eom-character /space-character / DBCS-whitespace / most-Unicode-class-
Zs)
tab-character = %x0009
eom-character = %x0019
space-character = %x0020
underscore = %x005F
DBCS-whitespace = %x3000
most-Unicode-class-Zs = <all members of Unicode class Zs which are not CP2-characters>
```

An implementation MAY limit the number of characters in an <extended-line>.

For ease of specification it is convenient to be able to explicitly refer to the point that immediately precedes the beginning of a logical line and the point immediately preceding the final <line-terminator> of a logical line. This is accomplished using <LINE-START> and <LINE-END> as terminal symbols of the VBA grammars. A <LINE-START> is defined to immediately precede each logical line and a <LINE-END> is defined as replacing the <line-terminator> at the end of each logical line:

```
module-body-lines = *logical-line
logical-line = LINE-START *extended-line LINE-END
```

When used in an ABNF rule definition <LINE-START> and <LINE-END> are used to indicated the required start or end of a <logical-line>.

3.3 Lexical Tokens

The syntax of VBA programs is most easily described in terms of *lexical tokens* rather than individual **Unicode** characters. In particular, the occurrence of whitespace or line-continuations between most syntactic elements is usually irrelevant to the syntactic grammar. The syntactic grammar is significantly simplified if it does not have to describe such possible whitespace occurrences. This is accomplished by using *lexical tokens* (also referred to simply as *tokens*) that abstract away whitespace as the terminal symbols of the syntactic grammar.

The lexical grammar defines the interpretation of a <module-body-lines> as a set of such *lexical tokens*.

The terminal elements of the lexical grammar are Unicode characters and the <LINE-START> and <LINE-END> elements. Generally any rule name of the lexical grammar that is written in all upper case characters is also a *lexical token* and terminal element of the VBA syntactic grammar. ABNF quoted literal text rules are also considered to be *lexical tokens* of the syntactic grammar. *Lexical tokens* encompass any white space characters that immediate precede them. Note that when used within the lexical grammar, quoted literal text rules are not treated as *tokens* and hence any preceding whitespace characters are significant.

3.3.1 Separator and Special Tokens

```
WS = 1*(WSC / line-continuation)
```

```
special-token = "," / "." / "!" /  "#" / "&" / "(" / ")" / "*" / "+" / "-" / "/" / ":" / ";"
/ "<" / "=" / ">" / "?" / "\" / "^"
NO-WS = <no whitespace characters allowed here>
NO-LINE-CONTINUATION = <a line-continuation is not allowed here>
EOL = [WS] LINE-END / single-quote comment-body
EOS = *(EOL  /  ":")  ;End Of Statement
single-quote = %x0027  ; '
comment-body = *(line-continuation / non-line-termination-character) LINE-END
```

<special-token> is used to identify single characters that have special meaning in the syntax of VBA programs. Because they are *lexical tokens (section 3.3)*, these characters can be preceded by white space characters that are ignored. Any occurrence of one of the quoted <special-token> elements as a grammar element within the syntactic grammar is a reference to the corresponding *token (section 3.3)*.

<NO-WS> is used as terminal element of the syntactic grammar to indicate that the *token* that immediately follows it *MUST NOT* be preceded by any white space characters. <NO-LINE-CONTINUATION> is used as terminal element of the syntactic grammar to indicate that the *token* that immediately follows it *MUST NOT* be preceded by white space that includes any <linecontinuation> sequences.

<WS> is used as a terminal element of the syntactic grammar to indicate that the *token* that immediately follows it *MUST* have been preceded by one or more white space characters.

<EOL> is used as element of the syntactic grammar to name the token that acts as an "end of statement" marker for statements that MUST be the only or last statement on a logical line.

<EOS> is used as a terminal element of the syntactic grammar to name the *token* that acts as an "end of statement" marker. In general, the end of statement is marked by either a <LINE-END> or a colon character. Any characters between a <single-quote> and a <LINE-END> are comment text that is ignored.

3.3.2 Number Tokens

```
INTEGER = integer-literal ["%" / "&" / "^"]
integer-literal = decimal-literal / octal-literal / hex-literal
decimal-literal = 1*decimal-digit
octal-literal = "&" [%x004F / %x006F] 1*octal-digit    ; & or &o or &O
hex-literal = "&" (%x0048 / %x0068) 1*hex-digit   ; &h or &H
octal-digit = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"
decimal-digit = octal-digit / "8" / "9"
hex-digit = decimal-digit / %x0041-0046 / %x0061-0066 ;A-F / a-f
```

Static Semantics

- The <decimal-digit>, <octal-digit>, and <hex-digit> sequences are interpreted as unsigned integer values represented respectively in decimal, octal, and hexadecimal notation.
- Each <INTEGER> has an associated constant *data value (section 2.1)*. The *data value*, *value type (section 2.1)* and *declared type (section 2.2)* of the constant is defined by the following table (if the Valid column shows No, this <INTEGER> is invalid):

Radix	Positive <INTEGER> value in the range	Type Suffix	Valid <INTEGER>?	Declared Type	Value Type	Signed Data Value
Decimal	0 ≤ n ≤ 32767	None	Yes	Integer	Integer	n

Radix	Positive <INTEGER> value in the range	Type Suffix	Valid <INTEGER>?	Declared Type	Value Type	Signed Data Value
Decimal	0 ≤ n ≤ 32767	"%"	Yes	Integer	Integer	n
Decimal	0 ≤ n ≤ 32767	"&"	Yes	Long	Integer	n
Decimal	0 ≤ n ≤ 32767	"^"	Yes	LongLong	Integer	n
Octal	0 ≤ n ≤ &o77777	None	Yes	Integer	Integer	n
Octal	0 ≤ n ≤ &o77777	"%"	Yes	Integer	Integer	n
Octal	0 ≤ n ≤ &o77777	"&"	Yes	Long	Integer	n
Octal	0 ≤ n ≤ &o77777	"^"	Yes	LongLong	Integer	n
Octal	&o100000 ≤ n ≤ &o177777	None	Yes	Integer	Integer	n – 65,536
Octal	&o100000 ≤ n ≤ &o177777	"%"	Yes	Integer	Integer	n – 65,536
Octal	&o100000 ≤ n ≤ &o177777	"&"	Yes	Long	Integer	n
Octal	&o100000 ≤ n ≤ &o177777	"^"	Yes	LongLong	Integer	n
Hex	0 ≤ n ≤ &H7FFF	None	Yes	Integer	Integer	n
Hex	0 ≤ n ≤ &H7FFF	"%"	Yes	Integer	Integer	n
Hex	0 ≤ n ≤ &H7FFF	"&"	Yes	Long	Integer	n
Hex	0 ≤ n ≤ &H7FFF	"^"	Yes	LongLong	Integer	n
Hex	&H8000 ≤ n ≤ &HFFFF	None	Yes	Integer	Integer	n – 65,536
Hex	&H8000 ≤ n ≤ &HFFFF	"%"	Yes	Integer	Integer	n – 65,536
Hex	&H8000 ≤ n ≤ &HFFFF	"&"	Yes	Long	Integer	n
Hex	&H8000 ≤ n ≤ &HFFFF	"^"	Yes	LongLong	Integer	n
Decimal	32768 ≤ n ≤ 2147483647	None	Yes	Long	Long	n
Decimal	n ≥ 32768	"%"	No			
Decimal	32768 ≤ n ≤ 2147483647	"&"	Yes	Long	Long	n
Decimal	32768 ≤ n ≤ 2147483647	"^"	Yes	LongLong	Long	n
Decimal	n ≥ 2147483647	None	(see note 1)	Double	Double	n# (see note 1)
Decimal	n ≥ 2147483647	"&"	No			

Radix	Positive <INTEGER> value in the range	Type Suffix	Valid <INTEGER>?	Declared Type	Value Type	Signed Data Value
Octal	&o200000 ≤ n ≤ &o17777777777	None	Yes	Long	Long	n
Octal	&o200000 ≤ n ≤ &o17777777777	"%"	No			
Octal	&o200000 ≤ n ≤ &o17777777777	"&"	Yes	Long	Long	n
Octal	&o200000 ≤ n ≤ &o17777777777	"^"	Yes	LongLong	Long	n
Octal	&o20000000000 ≤ n ≤ &o37777777777	None	Yes	Long	Long	n – 4,294,967,296
Octal	&o20000000000 ≤ n ≤ &o37777777777	"%"	No			
Octal	&o20000000000 ≤ n ≤ &o37777777777	"&"	Yes	Long	Long	n – 4,294,967,296
Octal	&o20000000000 ≤ n ≤ &o37777777777	"^"	Yes	LongLong	Long	n
Octal	n ≥ &o40000000000	None	No			
Octal	n ≥ &o40000000000	"%"	No			
Octal	n ≥ &o40000000000	"&"	No			
Hex	&H8000 ≤ n ≤ &H7FFFFFFF	None	Yes	Long	Long	n
Hex	&H8000 ≤ n ≤ &H7FFFFFFF	"%"	No			
Hex	&H8000 ≤ n ≤ &H7FFFFFFF	"&"	Yes	Long	Long	n
Hex	&H8000 ≤ n ≤ &H7FFFFFFF	"^"	Yes	LongLong	Long	n
Hex	&H80000000 ≤ n ≤ &H7FFFFFFF	None	Yes	Long	Long	n – 4,294,967,296
Hex	&H80000000 ≤ n ≤ &H7FFFFFFF	"%"	No			
Hex	&H80000000 ≤ n ≤ &H7FFFFFFF	"&"	Yes	Long	Long	n – 4,294,967,296
Hex	&H80000000 ≤ n ≤	"^"	Yes	LongLong	Long	n

Radix	Positive <INTEGER> value in the range	Type Suffix	Valid <INTEGER>?	Declared Type	Value Type	Signed Data Value
	&H7FFFFFFF					
Hex	n ≥ &H100000000	None	No			
Hex	n ≥ &H100000000	"%"	No			
Hex	n ≥ &H100000000	"&"	No			
Decimal	2147483648 ≤ n ≤ 9223372036854775807	"^"	Yes	LongLong	LongLong	n
Decimal	n ≥ 9223372036854775808	"^"				
Octal	&o40000000000 ≤ n ≤ &o1777777777777777777	"^"	Yes	LongLong	LongLong	n - 2^{32}
Octal	n ≥ &o2000000000000000000000	Any	No			
Hex	&H100000000 ≤ n ≤ &HFFFFFFFFFFFFFFFF	"^"	Yes	LongLong	LongLong	n - 2^{32}
Hex	n ≥ &H10000000000000000	Any	No			

- It is statically invalid for a literal to have the declared type LongLong in an implementation that does not support 64-bit arithmetic.

```
FLOAT = (floating-point-literal [floating-point-type-suffix] ) / (decimal-literal floating-
point-type-suffix)
floating-point-literal = (integer-digits exponent) / (integer-digits "." [fractional-digits]
[exponent]) / ( "." fractional-digits [exponent])

integer-digits = decimal-literal
fractional-digits = decimal-literal
exponent = exponent-letter  [sign] decimal-literal
exponent-letter = %x0044 / %x0045 / %x0064 / %x0065   ; D / E / d / e
sign = "+" / "-"
floating-point-type-suffix = "!" / "#" / "@"
```

Static Semantics

- <FLOAT> *tokens* represent either binary floating point or currency *data values*. The <floatingpoint-type-suffix> designates the *declared type* and *value type* of the *data value* associated with the *token* according to the following table:

<floating-point-type-suffix>	Declared Type and Value Type
Not present	**Double**
!	**Single**
#	**Double**
@	**Currency**

- Let i equal the integer value of <integer-digits>, f be the integer value of <fractional-digits>, d be the number of digits in <fractional-digits>, and x be the signed integer value of <exponent>. A <floating-point-literal> then represents a mathematical real number, r, according to this formula:

  $$r = \left(i + f10^{-d}\right)10^{x}$$

- A <floating-point-literal> is invalid if its mathematical value is greater than the greatest mathematical value that can be represented using its *declared type*.

- If the *declared type* of <floating-point-literal> is **Currency**, the fractional part of r is rounded using *Banker's rounding (section 5.5.1.2.1.1)* to 4 significant digits.

3.3.3 Date Tokens

```
date-or-time = (date-value 1*WSC time-value) / date-value / time-value

date-value = left-date-value date-separator  middle-date-value [date-separator right-date-
value]
left-date-value = decimal-literal / month-name
middle-date-value = decimal-literal / month-name
right-date-value = decimal-literal / month-name
date-separator = 1*WSC / (*WSC ("/" / "-" / ",") *WSC)

month-name = English-month-name / English-month-abbreviation
English-month-name = "january" / "february" / "march" / "april" / "may" / "june" / "august" /
"september" / "october" / "november" / "december"
English-month-abbreviation = "jan" / "feb" / "mar" / "apr" / "jun" / "jul" / "aug" / "sep" /
"oct" / "nov" / "dec"

time-value = (hour-value ampm) / (hour-value time-separator minute-value [time-separator
second-value] [ampm])
hour-value = decimal-literal
minute-value = decimal-literal
second-value = decimal-literal
time-separator = *WSC (":" / ".") *WSC
ampm = *WSC ("am" / "pm" / "a" / "p")
```

Static Semantics

- A <DATE> *token (section 3.3)* has an associated *data value (section 2.1)* of *value type (section 2.1)* and *declared type (section 2.2)* **Date**.

- The numeric data value of a <DATE> *token* is the sum of its *specified date* and its *specified time*.

- If a <date-or-time> does not include a <time-value> its *specified time* is determined as if a <time-value> consisting of the characters "00:00:00" was present.

- If a <date-or-time> does not include a <date-value> its *specified date* is determined as if a <date-value> consisting of the characters "1899/12/30" was present.

- At most one of <left-date-value>, <middle-date-value>, and <right-date-value> can be a <month-name>.
- Given that L is the *data value* of <left-date-value>, M is the *data value* of <middle-date-value>, and R is the *data value* of <right-date-value> if it is present. L, M, and R are interpreted as a calendar date as follows:
  - Let
    $$LegalMonth(x) = \begin{cases} true, 0 \le x \le 12 \\ false, otherwise \end{cases}$$
    $$LegalDay(month, day, year) = \begin{cases} false \begin{cases} year < 0 \; or \; year > 32767, or \\ LegalMonth(month) \; is \; false, or \\ day \; is \; not \; a \; valid \; day \; for \; the \; specified \; month \; and \; year \end{cases} \\ true, otherwise \end{cases}$$
  - Let
  - Let CY be an implementation-defined default year.
  - Let
    $$Year(x) = \begin{cases} x + 2000, 0 \le x \le 29 \\ x + 1900, 30 \le x \le 99 \\ x, otherwise \end{cases}$$
  - If L and M are numbers and R is not present:
    - If *LegalMonth(L)* and *LegalDay(L,M,CY)* then L is the month, M is the day, and the year is CY
    - Else if *LegalMonth(M)* and *LegalDay(M,L,CY)* then M is the month, L is the day, and the year is CY
    - Else if *LegalMonth(L)* then L is the month, the day is 1, and the year is M
    - Else if *LegalMonth(M)* then M is the month, the day is 1, and the year is L
    - Otherwise, the <date-value> is not valid.
  - If L, M, and R are numbers:
    - If *LegalMonth(L)* and *LegalDay(L,M,Year(R))* then L is the month, M is the day, and *Year(R)* is the year
    - Else if *LegalMonth(M)* and *LegalDay(M,R,Year(L))* then M is the month, R is the day, and *Year(L)* is the year
    - Else if *LegalMonth(M)* and *LegalDay(M,L,Year(R))* then M is the month, L is the day, and *Year(R)* is the year
    - Otherwise, the <date-value> is not valid.
  - If either L or M is not a number and R is not present:
    - Let N be the value of whichever of L or M is a number.
    - Let M be the value in the range 1 to 12 corresponding to the month name or abbreviation that is the value of whichever of L or M is not a number.

        - If *LegalDay(M,N,CY)* then M is the month, N is the day, and the year is CY
        - Otherwise, M is the month, 1 is the day, and the year is *Year(N)*.
    - Otherwise, R is present and one of L, M, and R is not a number:
        - Let M be the value in the range 1 to 12 corresponding to the month name or abbreviation that is the value of whichever of L, M, or R is not a number.
        - Let N1 and N1 be the numeric values of which every of L, M, or R are numbers.
        - If *LegalDay(M,N1,Year(N2)* then M is the month, N1 is the day, and *Year(N2)* is the year
        - If *LegalDay(M,N2,Year(N1)* then M is the month, N2 is the day, and *Year(N1)* is the year
        - Otherwise, the <date-value> is not valid.
- A <decimal-literal> that is an element of an <hour-value> MUST have an integer value in the inclusive range of 0 to 23.
- A <decimal-literal> that is an element of an <minute-value> MUST have an integer value in the inclusive range of 0 to 59.
- A <decimal-literal> that is an element of an <second-value> MUST have an integer value in the inclusive range of 0 to 59
- If <time-value> includes an <ampm> element that consists of "pm" or "p" and the <hour-value> has an integer value in the inclusive range of 0 to 11 then the <hour-value> is used as if its integer value was 12 greater than its actual integer value.
- A <ampm> element has no significance if the <hour-value> is greater than 12.
- If <time-value> includes an <ampm> element that consists of "am" or "a" and the <hour-value> is the integer value 12, then the <hour-value> is used as if its integer value was 0.
- If a <time-value> does not include a <minute-value> it is as if there was a <minute-value> whose integer value was 0.
- If a <time-value> does not include a <second-value> it is as if there was a <second-value> whose integer value was 0.
- Let h be the integer value of the <hour-value> element of a <time-value>, let m be the integer value of the <minute-value> element of that <time-value>, and let s be the integer value of the <second-value> of that <time-value>. The specified time of the <time-value> is defined by the formula (3600h+60m+s)/86400.

3.3.4 String Tokens

```
STRING = double-quote *string-character (double-quote /  line-continuation / LINE-END)
double-quote = %x0022  ; "
string-character = NO-LINE-CONTINUATION ((double-quote double-quote)  /  non-line-
termination-character)
```

Static Semantics

- A <STRING> *token (section 3.3)* has an associated *data value (section 2.1)* of *value type (section 2.1)* and *declared type (section 2.2)* **String**.

- The length of the associated string *data value* is the number of <string-character> elements that comprise the <STRING>
- The *data value* consists of the sequence of implementation-defined encoded characters corresponding to the <string-character> elements in left to right order where the left-most <string-character> element defines the first element of the sequence and the right-most <string-character> element defines the last character of the sequence.
- A <STRING> *token* is invalid if any <string-character> element does not have an encoding in the in the implementation-defined character set.
- A sequence of two <double-quote> characters represents a single occurrence of the character U+0022 within the *data value*.
- If there are no <string-character> elements, the *data value* is the zero length empty string.
- If a <STRING> ends in a <line-continuation> element, the final character of the associated *data value* is the right-most character preceding the <line-continuation> that is not a <WSC>.
- If a <STRING> ends in a <LINE-END> element, the final character of the associated *data value* is the right-most character preceding the <LINE-END> that is not a <line-terminator>.

3.3.5 Identifier Tokens

```
lex-identifier = Latin-identifier / codepage-identifier / Japanese-identifier / Korean-
identifier / simplified-Chinese-identifier / traditional-Chinese-identifier

Latin-identifier = first-Latin-identifier-character *subsequent-Latin-identifier-character
first-Latin-identifier-character = (%x0041-005A / %x0061-007A) ; A-Z / a-z
subsequent-Latin-identifier-character = first-Latin-identifier-character / decimal-digit /
%x5F    ; underscore
```

Static Semantics

- Upper and lowercase Latin characters are considered equivalent in VBA identifiers. Two identifiers that differ only in the case of corresponding <first-Latin-identifier-character> characters are considered to be the same identifier.
- Implementations MUST support <Latin-identifier>. Implementations MAY support one or more of the other identifier forms and if so MAY restrict the combined use of such identifier forms.

3.3.5.1 Non-Latin Identifiers

```
Japanese-identifier = first-Japanese-identifier-character *subsequent-Japanese-identifier-
character
first-Japanese-identifier-character = (first-Latin-identifier-character / CP932-initial-
character)
subsequent-Japanese-identifier-character = (subsequent-Latin-identifier-character / CP932-
subsequent-character)
CP932-initial-character = < character ranges specified in section 3.3.5.1.1>
CP932-subsequent-character = < character ranges specified in section 3.3.5.1.1>

Korean-identifier = first-Korean-identifier-character *subsequent Korean-identifier-character
first-Korean-identifier-character = (first-Latin-identifier-character / CP949-initial-
character )
subsequent-Korean-identifier-character = (subsequent-Latin-identifier-character / CP949-
subsequent-character)
CP949-initial-character = < character ranges specified in section 3.3.5.1.2>
CP949-subsequent-character = < character ranges specified in section 3.3.5.1.2>

simplified-Chinese-identifier = first-sChinese-identifier-character
         *subsequent-sChinese-identifier-character
```

```
first-sChinese-identifier-character = (first-Latin-identifier-character / CP936-initial-
character)
subsequent-sChinese-identifier-character = (subsequent-Latin-identifier-character / CP936-
subsequent-character)
CP936-initial-character = < character ranges specified in section 3.3.5.1.3>
CP936-subsequent-character = < character ranges specified in section 3.3.5.1.3>

traditional-Chinese-identifier = first-tChinese-identifier-character
                  *subsequent-tChinese-identifier-character
first-tChinese-identifier-character = (first-Latin-identifier-character / CP950-initial-
character)
subsequent-tChinese-identifier-character = (subsequent-Latin-identifier-character / CP950-
subsequent-character)
CP950-initial-character = < character ranges specified in section 3.3.5.1.4>
CP950-subsequent-character = < character ranges specified in section 3.3.5.1.4>

codepage-identifier = (first-Latin-identifier-character / CP2-character)
       *(subsequent-Latin-identifier-character / CP2-character)

CP2-character = <any Unicode character that has a mapping to the character range %x80-FF in a
Microsoft Windows supported code page>
```

VBA support for identifiers containing non-Latin ideographic characters was designed based upon characters code standards that predate the creation of **Unicode**. For this reason, non-Latin Identifiers are specified in terms of the Unicode characters corresponding to code points in these legacy standards rather than directly using similar Unicode characters classes.

Any Unicode character that corresponds to a character in a Microsoft Windows code page with a single byte code point in the range %x80-FF is a valid <CP2-characters>. The **code pages** defining such characters are Windows Codepages 874, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, and 1258. The definitions of these codepages and the mapping of individual codepage specific code points to Unicode code points are specified by files hosted at [UNICODE-BESTFIT] and explained by [UNICODE-README]. [CODEPG] provides an informative overview of the code pages code points and their mappings to the corresponding Unicode characters.

3.3.5.1.1 Japanese Identifiers

VBA support for identifiers containing Japanese characters is based upon Windows Codepage 932 [UNICODE-BESTFIT]. Japanese characters are encoded as both 8 bit single byte and 16 bit double byte characters with code points beginning at %x80. The **Unicode** equivalents of Windows Codepage 932 code points are specified by the file bestfit932.txt provided at [UNICODE-BESTFIT]. Many of the characters in the range %x80-FF are lead bytes that serve as the first byte of a 16 bit encoding of a code point. However, valid characters also occur within this range.

A <CP932-initial-character> can be any Unicode character that corresponds to a defined **code page** 932 character whose Windows Codepage 932 code point is greater than %x7F except for code points in the range %x80-FF that are lead bytes and except for the following code points that are explicitly excluded: %x8140, %x8143-8151,%x815E-8197,%x824f-8258.

A <CP932-subsequent-character> is defined identically to <CP932-initial-character> except that code points in the range are %x824f-8258 are not excluded.

3.3.5.1.2 Korean Identifiers

VBA support for identifiers containing Korean characters is based upon Windows Codepage 949 [UNICODE-BESTFIT]. Korean characters are encoded as 16 bit double byte characters with code points beginning at %x8141. The **Unicode** equivalents of Windows Codepage 949 code points are specified by the file bestfit949.txt provided at [UNICODE-BESTFIT]. All of the code points in the range %x81-FE are lead bytes that serve as the first byte of a 16 bit encoding of a code point.

A <CP949-initial-character> MAY be any Unicode character that corresponds to the following Windows Codepage 949 character code points: any defined 16-bit code point whose lead byte is less than %xA1 or greater than %xAF; any defined code point, regardless of its lead byte value, whose second bytes is less than %xA1 or greater than %xFE; code points in the range %xA3C1-A3DA; code points in the range %xA3E1-A3FA; code points in the range %xA4A1-A4FE.

A <CP949-subsequent-character> is defined identically to <CP949-initial-character> with the addition of code point %xA3DF and code points in the range %xA3B0-A3B9.

3.3.5.1.3 Simplified Chinese Identifiers

VBA support for identifiers containing Simplified Chinese characters is based upon Windows Codepage 936 [UNICODE-BESTFIT]. Simplified Chinese characters are encoded as 16 bit double byte characters with code points beginning at %x8140. The **Unicode** equivalents of Windows Codepage 936 code points are specified by the file bestfit936.txt provided at [UNICODE-BESTFIT].

A <CP936-initial-character> MAY be any Unicode character that corresponds to defined code points in the following ranges of Windows Codepage 936 code points: %xA3C1-A3DA; %xA3E1-A3FA; %xA1A2A1AA; %xA1AC-A1AD; %xA1B2-A1E6; %xA1E8-A1EF; %xA2B1-A2FC; %xA4A1-FE4F.

A <CP936-subsequent-character> is defined identically to <CP949-initial-character> with the addition of code point %xA3DF and code points in the range %xA3B0-A3B9.

3.3.5.1.4 Traditional Chinese Identifiers

VBA support for identifiers containing Traditional Chinese characters is based upon Windows Codepage 950 [UNICODE-BESTFIT]. Traditional Chinese characters are encoded as 16 bit double byte characters with code points beginning at %xA140. The **Unicode** equivalents of Windows Codepage 950 code points are specified by the file bestfit950.txt provided at [UNICODE-BESTFIT].

A <CP950-initial-character> MAY be any Unicode character that corresponds to defined code points in the following ranges of Windows Codepage 950 code points: %xA2CF-A2FE; %xA340-F9DD.

A <CP950-subsequent-character> is defined identically to <CP950-initial-character> with the addition of code point %xA1C5 and code points in the range %xA2AF-A2B8.

3.3.5.2 Reserved Identifiers and IDENTIFIER

```
reserved-identifier = statement-keyword / marker-keyword / operator-identifier /
     special-form / reserved-name / literal-identifier / rem-keyword /
     reserved-for-implementation-use / future-reserved

IDENTIFIER = <any lex-identifier that is not a reserved-identifier>
```

<reserved-identifier> designates all sequences of characters that conform to <Latin-identifier> but are reserved for special uses within the VBA language. *Keyword* is an alternative term meaning <reserved-identifier>. When a specific *keyword* needs to be named in prose sections of this specification the *keyword* is written using bold emphasis. Like all VBA identifiers, a <reserved-identifier> is case insensitive. A <reserved-identifier> is a *token (section 3.3)*. Any quoted occurrence of one of the <reserved-identifier> elements as a grammar element within the syntactic grammar is a reference to the corresponding *token*. The *token* element <IDENTIFIER> is used within the syntactic grammar to specify the occurrence of an identifier that is not a <reserved-identifier>

Static Semantics

- The *name value* of an <IDENTIFIER> is the text of its <lex-identifier>.

- The *name value* of a <reserved-identifier> *token* is the text of its <Latin-identifier>.
- *Two name values* are the same if they would compare equal using a case insensitive textual comparison.

<reserved-identifier> are categorized according to their usage by the following rules. Some of them have multiple uses and occur in multiple rules.

```
statement-keyword = "Call" / "Case" /"Close" / "Const"/ "Declare" / "DefBool" / "DefByte"
/ "DefCur" / "DefDate" / "DefDbl" / "DefInt" / "DefLng" / "DefLngLng" / "DefLngPtr" /
"DefObj" / "DefSng" / "DefStr" / "DefVar" / "Dim" / "Do" / "Else" / "ElseIf" / "End" /
"EndIf" /  "Enum" / "Erase" / "Event" / "Exit" / "For" / "Friend" / "Function" / "Get" /
"Global" / "GoSub" / "GoTo" / "If" / "Implements"/ "Input" / "Let" / "Lock" / "Loop" /
"LSet" / "Next" / "On" / "Open" / "Option" / "Print" / "Private" / "Public" / "Put" /
"RaiseEvent" / "ReDim" / "Resume" / "Return" / "RSet" / "Seek" / "Select" / "Set" /
"Static" / "Stop" / "Sub" / "Type" / "Unlock" / "Wend" / "While" / "With" / "Write"

rem-keyword = "Rem"
marker-keyword = "Any" / "As"/ "ByRef" / "ByVal "/"Case" / "Each" / "Else" /"In"/ "New" /
"Shared" / "Until" / "WithEvents" / "Write" / "Optional" / "ParamArray" / "Preserve" /
"Spc" / "Tab" / "Then" / "To"

operator-identifier = "AddressOf" / "And" / "Eqv" / "Imp" / "Is" / "Like" / "New" / "Mod"
/ "Not" / "Or" / "TypeOf" / "Xor"
```

A <statement-keyword> is a <reserved-identifier> that is the first syntactic item of a statement or declaration. A <marker-keyword> is a <reserved-identifier> that is used as part of the interior syntactic structure of a statement. An <operator-identifier> is a <reserved-identifier> that is used as an operator within expressions.

```
reserved-name = "Abs" / "CBool" / "CByte" / "CCur" / "CDate" / "CDbl" / "CDec" / "CInt" /
"CLng" / "CLngLng" / "CLngPtr" / "CSng" / "CStr" / "CVar" / "CVErr" / "Date" / "Debug" /
"DoEvents" / "Fix" / "Int" / "Len" / "LenB" / "Me" / "PSet" / "Scale" / "Sgn" / "String"

special-form = "Array" / "Circle" / "Input" / "InputB"  / "LBound" / "Scale" / "UBound"

reserved-type-identifier = "Boolean" / "Byte" / "Currency" / "Date" / "Double" /
"Integer" / "Long" / "LongLong" / "LongPtr" / "Single" / "String" / "Variant"


literal-identifier = boolean-literal-identifier / object-literal-identifier /
variant-literal-identifier
boolean-literal-identifier = "true" / "false"
object-literal-identifier = "nothing"
variant-literal-identifier = "empty" / "null"
```

A <reserved-name> is a <reserved-identifier> that is used within expressions as if it was a normal program defined *entity (section 2.2)*. A <special-form> is a <reserved-identifier> that is used in an expression as if it was a program defined procedure name but which has special syntactic rules for its argument. A <reserved-type-identifier> is used within a declaration to identify the specific *declared type (section 2.2)* of an *entity*.

A <literal-identifier> is a <reserved-identifier> that represents a specific distinguished *data value (section 2.1)*. A <boolean-literal-identifier> specifying "true" or "false" has a *declared type* of **Boolean** and a *data value* of **True** or **False**, respectively. An <object-literal-identifier> has a *declared type* of **Object** and the *data value* **Nothing**. A <variant-literal-identifier> specifying "empty" or "null" has a *declared type* of **Variant** and the *data value* **Empty** or **Null**, respectively.

```
reserved-for-implementation-use = "Attribute" / "LINEINPUT" / "VB Base" / "VB Control" /
"VB_Creatable" /  "VB_Customizable" / "VB_Description" / "VB_Exposed" / "VB_Ext_KEY " /
"VB_GlobalNameSpace" / "VB_HelpID" / "VB_Invoke_Func" / "VB_Invoke_Property " /
"VB_Invoke_PropertyPut" / "VB_Invoke_PropertyPutRefVB_MemberFlags" / "VB_Name" /
"VB_PredeclaredId" / "VB_ProcData" / "VB_TemplateDerived" / "VB_UserMemId" /
"VB VarDescription" / "VB VarHelpID" / "VB VarMemberFlags" / "VB VarProcData " /
"VB_VarUserMemId"

future-reserved = "CDecl" / "Decimal" / "DefDec"
```

A <reserved-for-implementation-use> is a <reserved-identifier> that currently has no defined meaning to the VBA language but is reserved for use by language implementers. A <future-reserved> is a <reserved-identifier> that currently has no defined meaning to the VBA language but is reserved for possible future extensions to the language.

3.3.5.3 Special Identifier Forms

```
FOREIGN-NAME = "[" foreign-identifier "]"
foreign-identifier = 1*non-line-termination-character
```

A <FOREIGN-NAME> is a *token (section 3.3)* that represents a text sequence that is used as if it was an identifier but which does not conform to the VBA rules for forming an identifier. Typically, a <FOREIGN-NAME> is used to refer to an *entity (section 2.2)* that is created using some programming language other than VBA.

Static Semantics

- The *name value (section 3.3.5.1)* of a <FOREIGN-NAME> is the text of its <foreign-identifier>.

```
   BUILTIN-TYPE = reserved-type-identifier /  ("[" reserved-type-identifier "]") /
"object" / "[object]"
```

In some VBA contexts, a <FOREIGN-NAME> whose *name value* is identical to a <reserved-type-identifier> can be used equivalently to that <reserved-type-identifier>. The identifier whose *name value* is "object" is not a <reserved-identifier> but is generally used as if it was a <reserved-type-identifier>.

Static Semantics

- The *name value* of a <BUILTIN-TYPE> is the text of its <reserved-type-identifier> element if it has one. Otherwise the *name value* is "object".
- The *declared type (section 2.2)* of a <BUILTIN-TYPE> element is the *declared type* whose name is the same as the *name value* of the <BUILTIN-TYPE>.

```
TYPED-NAME = IDENTIFIER  type-suffix

type-suffix = "%" / "&" / "^" / "!" / "#" / "@" / "$"
```

A <TYPED-NAME> is an <IDENTIFIER> that is immediately followed by a <type-suffix> with no intervening whitespace.

Static Semantics

- The *name value* of a <TYPED-NAME> is the *name value* of its <IDENTIFIER> elements.
- The *declared type* of a <TYPED-NAME> is defined by the following table:

<type-suffix>	Declared Type
%	**Integer**
&	**Long**
^	**LongLong**
!	**Single**
#	**Double**
@	**Currency**
$	**String**

3.4 Conditional Compilation

A module body can contain *logical lines (section 3.2)* that can be conditionally excluded from interpretation as part of the VBA program code defined by the *module (section 4.2)*. The *module body (section 4.2)* with such excluded lines logically removed is called the *preprocessed module body*. The *preprocessed module body* is determined by interpreting conditional compilation directives within tokenized <module-body-lines> conforming to the following grammar:

```
conditional-module-body = cc-block
cc-block = *(cc-const / cc-if-block / logical-line)
```

Static Semantics

- A <module-body-logical-structure> which does not conform to the rules of this grammar is not a valid VBA *module*.
- The <cc-block> that directly makes up a <conditional-module-body> is an *included block*.
- All <logical-line> lines that are immediate elements of an *included block* are included in the *preprocessed module body*.
- All <logical-line> lines that are immediate elements of an *excluded block (section 3.4.2)* are not included in the *preprocessed module body*.
- The relative ordering of the <logical-line> lines within the *preprocessed module body* is the same as the relative ordering of those lines within the original *module body*.

3.4.1 Conditional Compilation Const Directive

```
cc-const = LINE-START  "#"  "const" cc-var-lhs "=" cc-expression cc-eol
cc-var-lhs = name
cc-eol = [single-quote *non-line-termination-character] LINE-END
```

Static Semantics

- All <cc-const> lines are excluded from the *preprocessed module body (section 3.4)*.
- All <cc-const> directives are processed including those contained in *excluded blocks (section 3.4.2)*.
- If <cc-var-lhs> is a <TYPED-NAME> with a <type-suffix>, the <type-suffix> is ignored.
- The *name value (section 3.3.5.1)* of the <name> of a <cc-var-lhs> MUST be different for every <cc-var-lhs> (including those whose containing <cc-block> is an *excluded block*) within a <conditionalmodule-body>.
- The *data value (section 2.1)* of a <cc-expression> is the *constant value (section 5.6.16.2)* of the <cc-expression>.
- If *constant evaluation* of the <cc-expression> results in an evaluation error the content of the *preprocessed module body* is undefined.
- A <cc-const> defines a constant binding accessible to <cc-expression> elements of the containing *module*. The *bound name* is the *name value* of the <name> of the <cc-var-lhs> , the *declared type* of the *constant binding* is **Variant**, and the *data value* of the *constant binding* is the *data value* of the <cc-expression>.
- The *name value* of the <name> of a <cc-var-lhs> can be the same as a *bound name* of a project level conditional compilation constant. In that case, the constant binding defined by the <cc-const> element shadows the project level binding.

3.4.2 Conditional Compilation If Directives

```
cc-if-block = cc-if
    cc-block
   *cc-elseif-block
   [cc-else-block]
   cc-endif

cc-if = LINE-START  "#" "if" cc-expression "then" cc-eol

cc-elseif-block = cc-elseif cc-block
cc-elseif = LINE-START "#" "elseif" cc-expression "then" cc-eol

cc-else-block = cc-else cc-block
cc-else = LINE-START "#" "else" cc-eol

cc-endif = LINE-START "#" ("endif" / ("end" "if")) cc-eol
```

Static Semantics

- All of the constituent <cc-expression> elements of a <cc-if-block> MUST conform to the following rules, even if the <cc-if-block> is not contained within an *included block (section 3.4)*:
- The <cc-expression> within the <cc-if> and those within each <cc-elseif> are each evaluated.
- The *data values (section 2.1)* of the constituent <cc-expression> elements MUST all be **Let**-coercible to the **Boolean** *value type (section 2.1)*.
- If evaluation of any of the constituent *<cc-expression>* elements results in an evaluation error the content of the *preprocessed module body (section 3.4)* is undefined.

- If an <cc-if-block> is contained within an *included block* then at most one contained <cc-block> is selected as an *included block* according to the sequential application of these rules:

1. If the *evaluated value* of the <cc-expression> within the <cc-if> is a *true value*, the <cc-block> that immediate follows the <cc-if> is the *included block*.

2. If one or more of the <cc-expression> elements that are within a <cc-elseif> have an *evaluated value* that is a *true value* then the <cc-block> that immediately follows the first such <cc-elseif> is the *included block*.

3. If none of the evaluated <cc-expression> elements have a *true value* and a <cc-else-block> is present, the <cc-block> that is an element of the <cc-else-block> is the *included block*.

4. If none of the evaluated <*cc-expression*> have a *true value* and a <cc-else-block> is not present there is no *included block*.

- Any <cc-block> which is an immediate element of a <cc-if-block>, a <cc-elseif-block>, or a <cc-else-block> and which is not an *included block* is an *excluded block (section 3.4)*.

- All <cc-if>, <cc-elseif>, <cc-else>, and <cc-endif> lines are excluded from the *preprocessed module body*.

4 VBA Program Organization

A *VBA Environment* can be organized into a number of user-defined and *host application*-defined *projects (section 4.1)*. Each *project* is composed of one or more *modules (section 4.2)*.

4.1 Projects

A *project* is the unit in which VBA program code is defined and incorporated into a *VBA Environment*. Logically a *project* consists of a *project name*, a set of named modules, and an ordered list of *project references*. A *project reference* that occurs earlier in this list is said to have higher *reference precedence* than references that occur later in the list. The physical representation of a *project* and the mechanisms used for naming, storing, and accessing a project are implementation-defined.

A *project reference* specifies that a *project* accesses public *entities (section 2.2)* that are defined in another *project*. The mechanism for identifying a *project's* referenced projects is implementation defined.

There are three types of VBA *projects*: *source projects*, *host projects*, and *library projects*. *Source projects* are composed of VBA program code that exists in VBA Language source code form. A *library project* is a project that is defined in an implementation-defined manner that and can define all the same kinds of *entities* that a *source project* might define, except that it might not exist in VBA language source code form and might not have been implemented using the VBA language.

A *host project* is a *library project* that is introduced into a *VBA Environment* by the *host application*. The means of introduction is implementation dependent. The *public variables (section 5.2.3.1)*, constants, procedures, *classes (section 2.5)*, and UDTs defined by a *host project* are accessible to VBA *source projects* in the same *VBA Environment* as if the *host project* was a *source project*. An *open host project* is one to which additional *modules* can be added to it by agents other than the *host application*. The means of designating an *open host project* and of adding *modules* to one is implementation defined.

Static Semantics.

- A *project name* MUST be valid as an <IDENTIFIER>.
- A *project name* SHOULD NOT be "VBA"; this name is reserved for accessing the *VBA Standard Library (section 2.7.1)*.
- A *project name* SHOULD NOT be a <reserved-identifier>.
- The *project references* of a specific *project* MUST identify *projects* with distinct *project names*.
- It is implementation dependent whether or not a *source project* references a different *project* that has the same *project name* as the referencing *project*.

4.2 Modules

A *module* is the fundamental syntactic unit of VBA source code. The physical representation of a *module* is implementation dependent but logically a VBA *module* is a sequence of **Unicode** characters that conform to the VBA language grammars.

A module consists of two parts: a *module header* and a *module body*.

The *module header* is a set of *attributes* consisting of name/value pairs that specify the certain linguistic characteristics of a *module*. While a *module header* could be directly written by a human programmer, more typically a VBA implementation will mechanically generate module headers based upon the programmer's usage of implementation specific tools.

A *module body* consists of actual VBA Language source code and most typically is directly written by a human programmer.

VBA supports two kinds of *modules*, *procedural modules* and *class modules*, whose contents MUST conform to the grammar productions <procedural-module> and <class-module>, respectively:

```
procedural-module = LINE-START procedural-module-header EOS
                    LINE-START procedural-module-body
class-module = LINE-START class-module-header
               LINE-START class-module-body

procedural-module-header = attribute "VB_Name" attr-eq quoted-identifier attr-end

class-module-header = 1*class-attr

class-attr = attribute "VB_Name" attr-eq quoted-identifier attr-end
     /  attribute "VB_GlobalNameSpace" attr-eq "False" attr-end
     /  attribute "VB_Creatable" attr-eq "False" attr-end
     /  attribute "VB_PredeclaredId" attr-eq boolean-literal-identifier attr-end
     /  attribute "VB_Exposed" attr-eq boolean-literal-identifier attr-end
     /  attribute "VB_Customizable" attr-eq boolean-literal-identifier attr-end
attribute = LINE-START "Attribute"
attr-eq = "="
attr-end = LINE-END

quoted-identifier = double-quote NO-WS IDENTIFIER NO-WS double-quote
```

Static Semantics.

- The *name value (section 3.3.5.1)* of an <IDENTIFIER> that follows an <attribute> element is an *attribute name*.
- An element that follows an <attr-eq> element defines the *attribute value* for the *attribute name* that precedes the same <attr-eq>.
- The *attribute value* defined by a <quoted-identifier> is the *name value* of the contained identifier.
- The last <class-attr> for a specific *attribute name* within a given <class-module-header> provides the *attribute value* for its *attribute name*.
- If an <class-attr> for a specific *attribute name* does not exist in an <class-module-header> it is assumed that a default *attribute value* is associated with the *attribute name* according to the following table:

Attribute Name	Default Value
VB_Creatable	False
VB_Customizable	False
VB_Exposed	False
VB_GlobalNameSpace	False
VB_PredeclaredId	False

- The *module name* of a *module* is the *attribute value* of the module's VB_NAME attribute.
- A maximum length of a *module name* is 31 characters.
- A *module name* SHOULD NOT be a <reserved-identifier>.
- A *module's module name* might not be the same as the *project name (section 4.1)* of the *project* that contains the *module* or that of any *project (section 4.1)* referenced by the containing *project*.
- Every *module* contained in a *project* MUST have a distinct *module name*.
- Both the VB_GlobalNamespace and VB_Creatable *attributes* MUST have the *attribute value* "False" in a VBA *module* that is part of a VBA *source project (section 4.1)*. However *library projects (section 4.1)* can contain *modules* in which the *attributes values* of these *attributes* are "True".
- In addition to this section, the meaning of certain *attributes* and attribute combinations when used in the definition of *class modules* is defined in section 5.2.4.1. All other usage and meanings of *attributes* are implementation-dependent.

4.2.1 Module Extensibility

An *open host project (section 4.1)* can include *extensible modules*. *Extensible modules* are *modules (section 4.2)* that can be extended by identically named externally provided *extension modules* that are added to the host project. An *extension module* is a *module* that defines additional *variables (section 2.3)*, constants, procedures, and UDT *entities (section 2.2)*. The additional extension module *entities* behave as if they were directly defined within the corresponding *extensible module*. Note that this means *extensible modules* can define **WithEvents** variables which can then be the target of event handler procedures in an *extension module*.

The mechanisms by which *extension modules* can be added to a *host project (section 4.1)* are implementation-defined.

Static Semantics.

- The *module name (section 4.2)* of an *extension module* MUST be identical to that of the *extensible module* it is extending.
- An *extension module* can't define or redefine any *variables*, constants, procedures, enums, or UDTs that are already defined in its corresponding *extensible module*. The same name conflict rules apply as if the *extension module* elements were physically part of the *module body (section 4.2)* of the corresponding *extensible module*.
- Option directives contained in an *extension module* only apply to the *extension module* and not to the corresponding *extensible module*.
- It is implementation defined whether or not more than one *extension module* might exist within an *extensible project* for a specific *extensible module*.

5 Module Bodies

Module bodies (section 4.2) contain source code written using the syntax of the VBA programming language, as defined in this specification. This chapter defines the valid syntax, static semantic rules, and runtime semantics of *module bodies*.

Syntax is described using an ABNF [RFC4234] grammar incorporating terminal symbols defined in section 3. Except for where it explicitly identifies <LINE-START> and <LINE-END> elements this grammar ignores the physical line structure of files containing the source code of *module bodies*. The grammar also ignores conditional compilation directives and conditionally excluded sources code as described in section 3.4. This grammar applied to the *preprocessed module body (section 3.4)*; the source code is interpreted as if both lexical tokenization and conditional compilation preprocessing has been applied to it. This preprocessing assumption is made solely to simplify and clarify this specification. An implementation is not required to actually use such a processing model.

5.1 Module Body Structure

```
procedural-module-body = LINE-START  procedural-module-declaration-section
                         LINE-START  procedural-module-code-section

class-module-body = LINE-START  class-module-declaration-section
                    LINE-START  class-module-code-section
```

Both *procedural modules (section 4.2)* and *class modules (section 4.2)* have *module bodies (section 4.2)* that consist of two parts, a *declaration section (section 5.2)* and a *code section (section 5.3)*. Each section MUST occur as the first syntactic element of a physical line of its containing source file.

Throughout this specification the following common grammar rules are used for expressing various forms of *entity (section 2.2)* names:

```
unrestricted-name = name / reserved-identifier
name = untyped-name / TYPED-NAME
untyped-name = IDENTIFIER / FOREIGN-NAME
```

5.2 Module Declaration Section Structure

A *module's (section 4.2) declaration sections* consists of directive and declarations. Generally directives control the application of static semantic rules within the module. Declarations define named entities that exist within the runtime environment of a program.

```
procedural-module-declaration-section = [*(procedural-module-directive-element EOS) def-
directive]  *( procedural-module-declaration-element EOS)
class-module-declaration-section = [*(class-module-directive-element EOS) def-directive]
*(class-module-declaration-element EOS)
procedural-module-directive-element = common-option-directive / option-private-directive /
def-directive
procedural-module-declaration-element = common-module-declaration-element / global-variable-
declaration / public-const-declaration / public-type-declaration / public-external-procedure-
declaration / global-enum-declaration / common-option-directive / option-private-directive
```

```
class-module-directive-element = common-option-directive / def-directive / implements-
directive
class-module-declaration-element = common-module-declaration-element / event-declaration /
commonoption-directive / implements-directive
```

Static Semantics.

There are various restrictions on the number of occurrences and the relative ordering of directives and declarations within *module declaration sections*. These restrictions are specified as part of the definition of the specific individual directives and declarations elements.

5.2.1 Option Directives

Option *directives* are used to select alternative semantics for various language features.

```
common-option-directive = option-compare-directive /  option-base-directive / option-
explicit-directive  / rem-statement
```

Static Semantics.

- Each <common-option-directive> alternative can occur at most once in each <procedural-module-declaration-section> or <class-module-declaration-section>.
- An <option-private-directive> can occur at most once in each <procedural-module-declaration-section>.

5.2.1.1 Option Compare Directive

Option Compare directives determine the comparison rules used by *relational operators (section 5.6.9.5)* when applied to String *data values (section 2.1)* within a *module (section 4.2)*. This is known as the *comparison mode* of the *module*.

```
option-compare-directive = "Option"   "Compare"   ( "Binary" / "Text")
```

Static Semantics.

- If an <option-compare-directive> includes the **Binary** *keyword (section 3.3.5.1)* the *comparison mode* of the *module* is *binary-compare-mode*.
- If an <option-compare-directive> includes the **Text** *keyword* the *comparison mode* of the *module* is *text-compare-mode*.
- An <option-compare-directive> can occur at most once in a <procedural-module-declaration-section> or <class-module-declaration-section>.
- If a <procedural-module-declaration-section> or <class-module-declaration-section> does not contain a <option-compare-directive> the *comparison mode* for the *module* is *binary-compare-mode*.

5.2.1.2 Option Base Directive

Option Base directives set the default value used within a *module (section 4.2)* for *lower bound (section 2.1)* of all array dimensions that are not explicitly specified in a <lower-bound> of a <dim-spec>.

```
option-base-directive = "Option"   "Base"     INTEGER
```

Static Semantics.

- An <option-base-directive> can occur at most once in a <procedural-module-declaration-section> or <class-module-declaration-section>.
- If present an < option-base-directive> MUST come before the first occurrence of a <dim-spec> in the same <procedural-module-declaration-section> or <class-module-declaration-section>.
- The *data value (section 2.1)* of the <INTEGER> MUST be equal to either the integer *data value* 0 or the integer *data value* 1.
- The default *lower bound* for array dimensions in containing *module* is the *data value* of the <INTEGER> element.
- If a <procedural-module-declaration-section> or <class-module-declaration-section> does not contain an <option-base-directive> the default *lower bound* for array dimensions in the *module* is 0.

5.2.1.3 Option Explicit Directive

Option Explicit directives is used to set the *variable declaration mode* which controls whether or not *variables (section 2.3)* can be *implicitly declared (section 5.6.10)* within the containing *module (section 4.2)*.

```
option-explicit-directive = "Option"   "Explicit"
```

Static Semantics:

- If an <option-explicit-directive> is present within a *module*, the *variable declaration mode* of the *module* is *explicit-mode*.
- If an <option-explicit-directive> is not present within a *module*, the *variable declaration mode* of the *module* is *implicit-mode*.
- An <option-explicit-directive> can occur at most once in a <procedural-module-declaration-section> or <class-module-declaration-section>.
- If a <procedural-module-declaration-section> or <class-module-declaration-section> does not contain a <option-explicit-directive> the *variable declaration mode* for the *module* is *implicit-mode*.

5.2.1.4 Option Private Directive

Option Private directives control the accessibility of a *module (section 4.2)* to other *projects (section 4.1)*, as well as the meaning of public accessibility of **Public** *entities (section 2.2)* declared within the *module*.

```
option-private-directive = "Option"   "Private"   "Module"
```

Static Semantics:

- If a *procedural module (section 4.2)* contains an <option-private-directive>, the *module* itself is considered a *private module*, and is accessible only within the enclosing *project*.
- If a *procedural module* does not contain an <option-private-directive>, the module itself is considered a *public module*, and is accessible within the enclosing *project* and within any *projects* that reference the enclosing *project*.
- The effect of *module* accessibility on the accessibility of declarations within the *module* is described in the definitions of specific *module* declaration form within section 5.2.3.

5.2.2 Implicit Definition Directives

```
def-directive = def-type  letter-spec *( "," letter-spec)
letter-spec = single-letter /  universal-letter-range / letter-range

single-letter = IDENTIFIER   ; %x0041-005A / %x0061-007A

universal-letter-range = upper-case-A "-"upper-case-Z
upper-case-A = IDENTIFIER
upper-case-Z = IDENTIFIER

letter-range = first-letter  "-" last-letter
first-letter = IDENTIFIER
last-letter = IDENTIFIER

def-type = "DefBool" / "DefByte" / "DefCur" /  "DefDate" / "DefDbl" / "DefInt" / "DefLng" /
"DefLngLng" / "DefLngPtr" / "DefObj" / "DefSng" / "DefStr" / "DefVar"
```

Implicit Definition directives define the rules used within a *module (section 4.2)* for determining the *declared type (section 2.2)* of *implicitly typed entities (section 2.2)*. The *declared type* of such *entities* can be determined based upon the first character of its *name value (section 3.3.5.1)*. Implicit Definition directives define the mapping from such characters to *declared types*.

Static Semantics.

- The *name value* of the <IDENTIFIER> element of a <single-letter> MUST consist of a single upper or lower case alphabetic character (%x0041-005A or %x0061-007A).
- The *name value* of the <IDENTIFIER> element of a <upper-case-A> MUST consist of the single character "A" (%x0041).
- The *name value* of the <IDENTIFIER> element of a <upper-case-Z> MUST consist of the single character "Z" (%x005A).
- A <letter-spec> consisting of a <single-letter> defines the implicit *declared type* within the containing *module* of all <IDENTIFIER> tokens whose *name value* begins with the character that is the *name value* of the <IDENTIFIER> element of the <single-letter> .
- A <letter-spec> consisting of a <letter-range> defines the implicit *declared type* within the containing *module* of all *entities* with <IDENTIFIER> tokens whose *name values* begins with any of the characters in the contiguous span of characters whose first inclusive character is the *name value* of the <first-letter> <IDENTIFIER> element and whose last inclusive character is the *name*

value of the <last-letter> <IDENTIFIER> element. The span can be an ascending or descending span of characters and can consist of a single character.

- Within a <procedural-module-declaration-section> or <class-module-declaration-section>, no overlap is allowed among <letter-spec> productions.
- A <universal-letter-range> defines a single implicit *declared type* for every <IDENTIFIER> within a *module*, even those with a first character that would otherwise fall outside this range if it was interpreted as a <letter-range> from A-Z.

The declared type corresponding to each <def-type> is defined by the following table:

<def-type>	Declared Type
"DefBool"	Boolean
"DefByte"	Byte
"DefInt"	Integer
"DefLng"	Long
"DefLngLng"	LongLong
"DefLngPtr"	*LongPtr type alias*
"DefCur"	Currency
"DefSng"	Single
"DefDbl"	Double
"DefDate"	Date
"DefStr"	String
"DefObj"	Object reference
"DefVar"	Variant

If an *entity* is not explicitly typed and there is no applicable <def-type>, then the *declared type* of the *entity* is **Variant**.

5.2.3 Module Declarations

```
common-module-declaration-element = module-variable-declaration
common-module-declaration-element =/ private-const-declaration
common-module-declaration-element =/ private-type-declaration
common-module-declaration-element =/ enum-declaration
common-module-declaration-element =/ private-external-procedure-declaration
```

Any kind of *module (section 4.2)* can contain a <common-module-declaration-element>. All other declarations are specific to either <procedural-module> or <class-module>.

5.2.3.1 Module Variable Declaration Lists

module-variable-declaration = public-variable-declaration / private-variable-declaration

```
global-variable-declaration = "Global"  variable-declaration-list
public-variable-declaration = "Public" ["Shared"] module-variable-declaration-list
private-variable-declaration = ("Private" / "Dim") [ "Shared"] module-variable-declaration-
list

module-variable-declaration-list = (withevents-variable-dcl / variable-dcl)
      *( "," (withevents-variable-dcl  / variable-dcl) )
variable-declaration-list = variable-dcl *( "," variable-dcl )
```

<global-variable-declaration> and the optional **Shared** *keyword (section 3.3.5.1)* provides syntactic compatibility with other dialects of the Basic language and/or historic versions of VBA.

Static Semantics

- The occurrence of the *keyword* **Shared** has no meaning.

- Each *variable (section 2.3)* defined within a <module-variable-declaration> contained within the same *module (section 4.2)* MUST have a different *variable name (section 2.3)*.

- Each *variable* defined within a <module-variable-declaration> is a *module variable* and MUST have a *variable name* that is different from the name of any other *module variable*, module constant, enum member, or *procedure (section 2.4)* that is defined within the same *module*.

- A variable declaration that is part of a <global-variable-declaration> or <public-variable-declaration> declares a *public variable*. The *variable* is accessible within the enclosing *project (section 4.1)*. If the enclosing *module* is a *class module (section 4.2)* or is a *procedural module (section 4.2)* that is not a *private module (section 5.2.1.4)*, then the *variable* is also accessible within *projects* that reference the enclosing *project*.

- A variable declaration that is part of a <private-variable-declaration> declares a *private variable*. The *variable* is only accessible within the enclosing *module*.

- If a *variable* defined by a <public-variable-declaration> has a *variable name* that is the same as a *project name (section 4.1)* or a *module name (section 4.2)* then all references to the *variable name* MUST be module qualified unless they occur within the *module* that contains the <public-variable-declaration>

- A *variable* defined by a <module-variable-declaration> can have a *variable name* that is the same as the enum name of a <enum-declaration> defined in the same *module* but such a *variable* cannot be referenced using its *variable name* even if the *variable name* is module qualified.

- If a *variable* defined by a <public-variable-declaration> has a *variable name* that is the same as the enum name of a public <enum-declaration> in a different *module*, all references to the *variable name* MUST be module qualified unless they occur within the *module* that contains the <public-variable-declaration>.

- The *declared type (section 2.2)* of a *variable* defined by a <public-variable-declaration> in a <class-module-code-section> might not be a *UDT (section 2.1)* that is defined by a <private-type-declaration> or a private enum name.

- A <module-variable-declaration-list> that occurs in a *procedural module* MUST NOT include any <withevents-variable-dcl> elements.

Runtime Semantics.

- All *variables* defined by a <module-variable-declaration> that is an element of in a <procedural-module-declaration-section> have *module extent (section 2.3)*.
- All *variables* defined by a <module-variable-declaration> that is an element of in a <class-module-declaration-section> are *member (section 2.5) variables* of the *class (section 2.5)* and have *object extent (section 2.3)*. Each *instance (section 2.5)* of the class will contain a distinct corresponding *variable*.

5.2.3.1.1 Variable Declarations

```
variable-dcl = typed-variable-dcl / untyped-variable-dcl
typed-variable-dcl = TYPED-NAME [array-dim]
untyped-variable-dcl = IDENTIFIER  [array-clause / as-clause]
array-clause = array-dim [as-clause]
as-clause = as-auto-object / as-type
```

Static Semantics

- A <typed-variable-dcl> defines a *variable (section 2.3)* whose *variable name (section 2.3)* is the *name value (section 3.3.5.1)* of the <TYPED-NAME>.
- If the optional <array-dim> is not present the *declared type (section 2.2)* of the defined *variable* is the *declared type* of the <TYPED-NAME>.
- If the optional <array-dim> is present and does not include a <bounds-list> then the *declared type* of the defined *variable* is *resizable array (section 2.2)* with an *element type (section 2.1.1)* that is the *declared type* of the <TYPED-NAME>.
- If the optional <array-dim> is present and includes a <bounds-list> then the *declared type* of the defined *variable* is *fixed-size array (section 2.2)* with an *element type* that is the *declared type* of the <TYPED-NAME>. The number of dimensions and the *upper bound (section 2.1)* and *lower bound (section 2.1)* for each dimension is as defined by the <bounds-list>.
- An <untyped-variable-dcl> that includes an <as-clause> containing an <as-auto-object> element defines an *automatic instantiation variable (section 2.5.1)*. If the <untyped-variable-dcl> also includes an <array-dim> element then each *dependent variable (section 2.3.1)* of the defined array *variable* is an *automatic instantiation variable*.
- If the <untyped-variable-dcl> does not include an <as-clause> (either directly or as part of an <array-clause> this is an *implicitly typed (section 5.2.2)* declaration and its *implicit declared type (section 5.2.3.1.5)*. The following rules apply:
  - The *declared type* of a *variable* defined by an *implicitly typed* declaration that does not include an <array-clause> is the same as its *implicit declared type*.
  - The *declared type* of a *variable* defined by an *implicitly typed* declaration that includes an <array-clause> whose <array-dim> element does not contain a <bounds-list> is *resizable array* whose *declared element type* is the same as the *implicit declared type*.
  - The *declared type* of a *variable* defined by an *implicitly typed* declaration that includes an <array-clause> whose <array-dim> element contains a <bounds-list> is *fixed size array* with a *declared element type* is the same as the *implicit declared type*. The number of dimensions and the *upper bound* and *lower bound* for each dimension is as defined by the <bounds-list>.

- If the <untyped-variable-dcl> includes an <array-clause> containing an <as-clause> the following rules apply:
  - If the <array-dim> of the <array-clause> does not contain a <bounds-list> the *declared type* of the defined *variable* is *resizable array* with a *declared element type* as the specified type of the <as-clause>.
  - If the <array-dim> of the <array-clause> contains a <bounds-list> the *declared type* of the defined *variable* is *fixed size array* with a *declared element type* as the specified type of the <as-clause>. The number of *dimensions* and the *upper* and *lower bound* for each *dimension* is as defined by the <bounds-list>.
  - If the <as-clause> consists of an <as-auto-object> each *dependent variable* of the defined *variable* is an *automatic instantiations variable*.
- If the <untyped-variable-dcl> includes an <as-clause> but does not include an <array-clause> the following rules apply:
  - The declared type of the defined variable is the specified type of the <as-clause>.
  - If the <as-clause> consists of an <as-auto-object> the defined variable is an *automatic instantiations variable*.

5.2.3.1.2 WithEvents Variable Declarations

```
withevents-variable-dcl = "withevents" IDENTIFIER "as" class-type-name

class-type-name = defined-type-expression
```

Static Semantics

- A <withevents-variable-dcl> defines a *variable* whose *declared type* is the specified type of its <class-type-name element.
- The *specified type* of the <class-type-name> element MUST be a specific *class* that has at least one *event member*.
- The *specified type* of <class-type-name> element MUST NOT be the *class* defined by the *class modules* containing this declaration.
  - The *name value* of the <IDENTIFIER with an appended underscore character (**Unicode** u+005F) is an *event handler name prefix* for the *class module* containing this declaration.
- The *specified type* of a <class-type-name> is the *declared type* referenced by its <defined-type-expression.

5.2.3.1.3 Array Dimensions and Bounds

```
array-dim = "(" [bounds-list] ")"
bounds-list = dim-spec *("," dim-spec)
dim-spec = [lower-bound] upper-bound
lower-bound = constant-expression  "to"
upper-bound = constant-expression
```

Static Semantics

- An <array-dim> that does not have a <bounds-list> designates a resizable array.
- A <bounds-list> contains at most 60 <dim-spec> elements.
- An <array-dim> with a <bounds-list> designates a *fixed-size array* with a number of *dimensions* equal to the number of <dim-spec> elements in the <bounds-list>.
- The <constant-expression> in an <upper-bound> or <lower-bound> MUST evaluate to a *data value* that is ***Let****-coercible* to the *declared type* Long.
- The *upper bound* of a *dimension* is specified by the Long *data value* of the <upper-bound> of the <dim-spec> that corresponds to the *dimension*.
- If the <lower-bound> is present, its <constant-expression> provides the *lower bound* Long *data value* for the corresponding *dimension*.
- If the <lower- bound> is not present the *lower bound* for the corresponding *dimension* is the default *lower bound* for the containing *module* as specified in section 5.2.1.2.
- For each *dimension*, the *lower bound* value MUST be less than or equal to the *upper bound* value.

5.2.3.1.4 Variable Type Declarations

A type specification determines the *specified type* of a declaration.

```
as-auto-object = "as" "new" class-type-name
as-type = "as" type-spec
type-spec = fixed-length-string-spec  / type-expression
fixed-length-string-spec = "string" "*" string-length
string-length = constant-name / INTEGER
constant-name = simple-name-expression
```

Static Semantics

- The *specified type* of an <as-auto-object> element is the *specified type* of its <class-type-name> element.
- The *specified type* of an <as-auto-object> element MUST be a named *class*.
- The *instancing mode* of the *specified type* of an <as-auto-object> MUST NOT be Public Not Creatable unless that type is defined in the same *project* as that which contains the module containing the <as-auto-object> element.
- The *specified type* of an <as-type> is the *specified type* of its <type-spec> element.
- The *specified type* of a <type-spec> is the *specified type* of its constituent element.
- The *specified type* of a <fixed-length-string-spec> is String*n where n is the *data value* of its <string-length> element.
- The *specified type* of a <type-expression> is the *declared type* referenced by the < type-expression>.
- A <constant-name> that is an element of a <string-length> MUST reference an explicitly-declared constant *data value* that is ***Let****-coercible* to the *declared type* **Long**.

- The *data value* of a <string-length> element is the *data value* of its <INTEGER> element or the *data value* referenced by its <constant-name> *Let-coerced* to *declared type* **Long**.
- The *data value* of a <string-length> element MUST be less than or equal to 65,526.
- The <simple-name-expression> element of <constant-name> MUST be *classified* as a *value*.

5.2.3.1.5 Implicit Type Determination

An <IDENTIFIER> that is not explicitly associated with a *declared type* via either a <type-spec> or a <type-suffix> might be implicitly associated with a *declared type*. The implicit *declared type* of such a name is defined as follows:

- If the first letter of the *name value* of the <IDENTIFIER> has is in the character span of a <letter-spec> that is part of a <def-directive> within the *module* containing the <IDENTIFIER> then its *declared type* is as specified in section 5.2.2.
- Otherwise its implicit *declared type* is **Variant.**

5.2.3.2 Const Declarations

```
public-const-declaration = ("Global" / "Public")  module-const-declaration
private-const-declaration = ["Private"] module-const-declaration
module-const-declaration = const-declaration

const-declaration = "Const"  const-item-list
const-item-list = const-item *[ "," const-item]
const-item = typed-name-const-item / untyped-name-const-item

typed-name-const-item = TYPED-NAME "=" constant-expression
untyped-name-const-item = IDENTIFIER [const-as-clause] "=" constant-expression

const-as-clause = "as" BUILTIN-TYPE
```

Static Semantics

- The <BUILTIN-TYPE> element of an <const-as-clause> might not be "object" or "[object]".
- Each *constant* defined within a <module-const-declaration> contained within the same module MUST have a different name.
- Each constant defined within a <module-const-declaration> MUST have a constant name that is different from any other module variable name, module constant name, enum member name, or procedure name that is defined within the same module.
- A constant declaration that is part of a <public-const-declaration> declares a *public constant*. The constant is accessible within the enclosing project. If the enclosing module is a procedural module that is not a private module, then the constant is also accessible within projects that reference the enclosing project.
- A constant declaration that is part of a <private-const-declaration> declares a *private constant*. The constant is accessible within the enclosing module.
- If a constant defined by a <public-const-declaration> has a constant name that is the same as the name of a project or name of a module then all references to the variable name MUST be module qualified unless they occur within the module that contains the <public-const-declaration>

- A constant defined by a <module-const-declaration> can have a constant name that is the same as the enum name of a <enum-declaration> defined in the same module but such a constant cannot be referenced using its constant name even if the constant name is module qualified.

- If a constant defined by a <public-const-declaration> has a constant name that is the same as the enum name of a public <enum-declaration> in a different module, all references to the constant name MUST be module qualified unless they occur within the module that contains the <public-const-declaration>.

- A <typed-name-const-item> defines a constant whose name is the name value of its <TYPED-NAME> element and whose declared type is the declared type corresponding to the <type-suffix> of the <TYPED-NAME> as specified in section 3.3.5.3.

- A <untyped-name-const-item> defines a constant whose name is the name value of its <IDENTIFIER> element.

- If an <untyped-name-const-item> does not include a <const-as-clause>, the declared type of the constant is the same as the declared type of its <constant-expression> element. Otherwise, the constant's declared type is the declared type of the <BUILTIN-TYPE> element of the <const-as-clause>.

- Any <constant-expression> used within a <const-item> might not reference functions, even the intrinsic functions normally permitted within a <constant-expression>.

- The data value of the <constant-expression> element in a <const-item> MUST be let-coercible to the declared type of the constant defined by that <const-item>

- The constant binding of a constant defined by a <const-item> is the data value of the <constant-expression> **Let**-coerced to the declared type of the constant.

5.2.3.3 User Defined Type Declarations

```
public-type-declaration = ["global" / "public"]  udt-declaration
private-type-declaration = "private" udt-declaration
udt-declaration = "type" untyped-name EOS udt-member-list EOS "end" "type"
udt-member-list = udt-element *[EOS udt-element]
udt-element = rem-statement / udt-member
udt-member = reserved-name-member-dcl / untyped-name-member-dcl
untyped-name-member-dcl = IDENTIFIER optional-array-clause
reserved-name-member-dcl = reserved-member-name as-clause
optional-array-clause = [array-dim] as-clause

reserved-member-name = statement-keyword / marker-keyword / operator-identifier / special-
form / reserved-name / literal-identifier / reserved-for-implementation-use / future-reserved
```

Static Semantics

- The *UDT name* of the containing <udt-declaration> is the name value of the <untyped-name> that follows the **Type** *keyword (section 3.3.5.1)*.

- Each <udt-declaration>defines a unique declared type and unique UDT value type each of which is identified by the UDT name.

- A UDT declaration that is part of a <public-const-declaration> declares a *public UDT*. The UDT is accessible within the enclosing project. If the enclosing module is a procedural module that is not a private module, then the UDT is also accessible within projects that reference the enclosing project.

- A UDT declaration that is part of a <private-const-declaration> declares a *private UDT*. The UDT is accessible within the enclosing module.
- If an <udt-declaration> is an element of a <private-type-declaration> its UDT name cannot be the same as the enum name of any <enum-declaration> or the UDT name of any other <udt-declaration> within the same module.
- If an <udt-declaration> is an element of a <public-type-declaration> its UDT name cannot be the same as the enum name of a public <enum-declaration> or the UDT name of any <public-type-declaration> within any module of the project that contains it.
- If an <udt-declaration> is an element of a <public-type-declaration> its UDT name cannot be the same as the name of any project or library within the current *VBA Environment* or the same name as any module within the project that contains the <udt-declaration>.
- The name value of a <reserved-member-name> is the text of its reserved identifier name. ☐ At least one <udt-element> in a <udt-member-list> MUST consist of a <udt-member>.
- If a <udt-member> is an <untyped-name-member-dcl> its udt member name is the name value of the <IDENTIFIER> element of the <untyped-name-member-dcl>.
- If a <udt-member> is a <reserved-name-member-dcl> its udt member name is the name value of the <reserved-member-name> element of the <reserved-name-member-dcl>.
- Each <udt-member> within a <udt-member-list> MUST have a different udt member name.
- Each <udt-member> defines a named element of the UDT value type identified by the UDT name of the containing <udt-declaration>.
- Each <udt-member> defines a named element of the UDT value type and declared type identified by the UDT name of the containing <udt-declaration>.
- The declared type of the UDT element defined by a <udt-member> is defined as follows:
  - If the <udt-member> contains an <array-dim> that does not contain a <bounds-list>, then the declared type of the UDT element is resizable array with a declared element type is the specified type of the <as-clause> contained in the <udt-member>.
  - If the <udt-member> contains an <array-dim> that contains a <bounds-list>, then the declared type of the UDT element is fixed size array whose declared element type is the specified type of the <as-clause> contained in the <udt-member>. The number of dimensions and the upper and lower bound for each dimension is as defined by the <bounds-list>.
  - Otherwise the declared type of the UDT element is the specified type of the <as-clause>.
- If a <udt-member> contains an <as-clause> that consists of an <as-auto-object> then the corresponding dependent variable (or each dependent variable of an array variable) of any variable whose declared type is the UDT name of the containing <udt-declaration> is an automatic instantiations variable.

5.2.3.4 Enum Declarations

```
global-enum-declaration = "global" enum-declaration
public-enum-declaration = ["public"] enum-declaration
private-enum-declaration = "private" enum-declaration
enum-declaration = "enum" untyped-name EOS member-list EOS "end" "enum"
enum-member-list = enum-element *[EOS enum-element]
enum-element = rem-statement / enum-member
enum-member = untyped-name [ "=" constant-expression]
```

<global-enum-declaration> provides syntactic compatibility with other dialects of the Basic language and historic versions of VBA.

Static Semantics.

- The *name value* of the <untyped-name> that follows the **Enum** *keyword (*section 3.3.5.1*)* is the *enum name* of the containing <enum-declaration>.
- An Enum declaration that is part of a <global-variable-declaration> or <public-enum-declaration> declares a *public Enum type*. The Enum type and its Enum members are accessible within the enclosing project. If the enclosing module is a class module or a procedural module that is not a private module, then the Enum type and its Enum members are also accessible within projects that reference the enclosing project.
- An Enum declaration that is part of a <private-enum-declaration> declares a *private Enum type*. The Enum type and its enum members are accessible within the enclosing module.
- The enum name of a <private-enum-declaration> cannot be the same as the enum name of any other <enum-declaration> or as the UDT name of a <udt-declaration> within the same module.
- The enum name of a <public-enum-declaration> cannot be the same as the enum name of any other public <enum-declaration> or the UDT name of any public <udt-declaration> within any module of the project that contains it.
- The enum name of a <public-enum-declaration> cannot be the same as the name of any project or library within the current *VBA Environment* or the same name as any module within the project that contains the <enum-declaration>.
- At least one <enum-element> in an <enum-member-list> MUST consist of a <enum-member>.
- The enum member name of an <enum-member> is the *name value* of its <untyped-name>.
- Each <enum-member> within a <enum-member-list> MUST have a different enum member name.
- An enum member name might not be the same as any variable name, or constant name that is defined within the same module.
- If an <enum-member> contains a <constant-expression>, the data value of the <constant-expression> MUST be coercible to value type Long.
- The <constant-expression> of an <enum-member> might not contain a reference to the enum member name of that <enum-member>.
- The <constant-expression> of an <enum-member> might not contain a reference to the enum member name of any <enum-member> that it precedes in its containing <enum-member-list>
- The <constant-expression> of an <enum-member> might not contain a reference to the enum member name of any <enum-member> of any <enum-declaration> that it precedes in the containing module declaration section.
- If an <enum-member> contains a <constant-expression>, the data value of the <enum-member> is the data value of its <constant-expression> coerced to value type Long. If an <enum-member> does not contain a <constant-expression> and it is the first element of a <enum-member-list> its data value is 0. If an <enum-member> does not contain a <constant-expression> and is not the first element of a <enum-member-list> its data value is 1 greater than the data value of the preceding element of its containing <enum-member-list>.

- The declared type of a <enum-member> is Long.
- When an enum name (possibly qualified by a project) appears in an <as-type> clause of any declaration, the meaning is the same as if the enum name was replaced with the declared type Long.

5.2.3.5 External Procedure Declaration

public-external-procedure-declaration = ["public"] external-proc-dcl

```
private-external-procedure-declaration = "private" external-proc-dcl

external-proc-dcl = "declare" ["ptrsafe"] (external-sub / external-function)

external-sub = "sub" subroutine-name lib-info [procedure-parameters]
external-function = "function" function-name lib-info  [procedure-parameters] [function-
type]

lib-info = lib-clause [alias-clause]
lib-clause = "lib" STRING
alias-clause = "alias" STRING
```

Static Semantics.

- <public-external-procedure-declaration> elements and <private-external-procedure-declaration> elements are *external procedures*.
- <public-external-procedure-declaration> elements and <private-external-procedure-declaration> elements are procedure declarations and the static semantic rules for procedure declarations define in section 5.3.1 apply to them.
- An <external-sub> element is a function declaration and an <external-function> is a subroutine declaration.
- It is implementation-defined whether an external procedure name is interpreted in a case sensitive or case-insensitive manner.
- If the first character of the <STRING> element of an <alias-clause> is the character %x0023 ("#") the element is an *ordinal alias* and the remainder of the string MUST conform to the definition of the <integer-literal> rule of the lexical token grammar. The data value of the <integer-literal> MUST be in the range of 0 to 32,767.
- If the first character of the data value of the <STRING> element of an <alias-clause> is not the character %x0023 ("#"), the data value of the <STRING> element MUST conform to an implementation-defined syntax.
- An implementation MAY define additional restrictions on the parameter types, function type, parameter mechanisms, and the use of optional and ParamArray parameters in the declaration of external procedures.
- An implementation MAY define additional restrictions on external procedure declarations that do not specify the PtrSafe keyword.

Runtime Semantics

- When an external procedure is called, the data value of the <STRING> element of its <lib-clause> is used in an implementation-defined manner to identify a set of available procedures that are defined using implementation-defined means other than the VBA Language.
- When an external procedure is called, the data value of the <STRING> element of it optional <alias-clause> is used in an implementation-defined manner to select a procedure from the set of available procedure. If an <alias-clause> is not present the name value of the procedure name is used in an implementation-defined manner to select a procedure from the set of available procedure.
- An external procedure is invoked and arguments passed as if the external procedure was a procedure defined in the VBA language by a <subroutine-declaration> or <function-declaration> containing the <procedure-parameters> and <function-type> elements of the external procedure's <external-proc-dcl>.

5.2.3.6 Circular Module Dependencies

Static Semantics.

- Circular reference between modules that involving *Const Declarations (section 5.2.3.2)*, *Enum Declarations (*section 5.2.3.4*)*, *UDT Declarations (section 5.2.3.3)*, *Implements Directive (section 5.2.4.2)*, or *Event Declarations (section 5.2.4.3)* are not allowed.
- Any circular dependency among modules that includes any of these declaration forms is an illegal circularity, even if the dependency chain includes other forms of declaration.
- Circular dependency chains among modules that do not include any of these specific declaration forms are allowed.

5.2.4 Class Module Declarations

Class modules define named classes that can be referenced as declared types by other modules within a *VBA Environment*.

5.2.4.1 Non-Syntactic Class Characteristics

Some of the characteristic of classes are not defined within the <class-module-body> but are instead defined using module attribute values and possibly implementation-defined mechanisms.

The name of the class defined by this class module is the name of the class module itself.

5.2.4.1.1 Class Accessibility and Instancing

The ability to reference a class by its name is determined by the *accessibility* of its class definition. This accessibility is distinct from the ability to use the class name to create new instances of the class.

The accessibility and *instancing* characteristics of a class are determined by the module attributes on its class module declaration, as defined by the following table:

Instancing Mode	Meaning	Attribute Values
Private *(default)*	The class is accessible only within the enclosing project. Instances of the class can only be created by modules contained within the project that defines the class.	VB_Exposed=False VB_Creatable=False
Public Not Creatable	The class is accessible within the enclosing project and within projects that reference the enclosing project. Instances of the class can only be created by modules within the enclosing project. Modules in other projects can reference the class name as a declared type but can't instantiate the class using new or the CreateObject function.	VB_Exposed=True VB_Creatable=False
Public Creatable	The class is accessible within the enclosing project and within projects that reference the enclosing project. Any module that can access the class can create instances of it.	VB_Exposed=True VB_Creatable=True

An implementation MAY define additional instancing modes that apply to classes defined by library projects.

5.2.4.1.2 Default Instance Variables Static Semantics

- A class module has a *default instance variable* if its VB_PredeclaredId attribute or
- VB_GlobalNamespace attribute has the value "True". This default instance variable is created with module extent as if declared in a <module-variable-declaration> containing an <as-autoobject> element whose <class-type-name> was the name of the class.
- If this class module's VB_PredeclaredId attribute has the value "True", this default instance variable is given the name of the class as its name. It is invalid for this named variable to be the target of a **Set** assignment. Otherwise, if this class module's VB_PredeclaredId attribute does not have the value "True", this default instance variable has no publicly expressible name.
- If this class module's VB_GlobalNamespace attribute has the value "True", the class module is considered a *global class module*, allowing simple name access to its default instance's members as specified in section 5.6.10.
- Note that if the VB_PredeclaredId and VB_GlobalNamespace attributes both have the value "True", the same default instance variable is shared by the semantics of both attributes.

5.2.4.2 Implements Directive

```
implements-directive = "Implements" class-type-name
```

Static Semantics.

- An <implements-directive> cannot occur within an extension module.
- The specified class of the <class-type-name> is called the *interface class*.
- The interface class can't be the class defined by the class module containing the <implements-directive>
- A specific class can't be identified as an interface class in more than one <implements-directive> in the same class module.
- The unqualified class names of all the interface classes in the same class module MUST be distinct from each other.
- The name value of the interface class's class name with an appended underscore character (**Unicode** u+005F) is an *implemented interface name prefix* within the class module containing this directive.
- If a class module contains more than one <implements-directive> then none of its implemented interface name prefixes can be occur as the initial text of any other of its implemented name prefix.
- A class can't be used as an interface class if the names of any of its public variable or method methods contain an underscore character (Unicode u+005F).
- A class module containing an <implements-directive> MUST contain an implemented name declaration corresponding to each public method declaration contained within the interface class' class module.
- A class module containing an <implements-directive> MUST contain an implemented name declaration corresponding to each public variable declaration contained within the interface class' class module. The set of required implemented name declarations depends upon of the declared type of the public variable as follows:
  - If the declared type of the variable is Variant there MUST be three corresponding implemented name declarations including a <property-get-declaration> and a <property-lhs-declaration>.
  - If the declared type of the variable is Object or a named class there MUST be two corresponding implemented name declarations including a <property-get-declaration> and a <property-lhs-declaration>.
  - If the declared type of the variable is anything else, there MUST be two corresponding implemented name declarations including a <property-get-declaration> and a <property-lhs-declaration>.

5.2.4.3 Event Declaration

```
event-declaration = ["Public"]
"Event" IDENTIFIER [event-parameter-list]
event-parameter-list = "(" [positional-parameters] ")"
```

Static Semantics

- An <event-declaration> defines an *event member* of the class defined by the enclosing class module.
- An <event-declaration> that does not begin with the *keyword (section 3.3.5.1)* **Public** has the same meaning as if the *keyword* **Public** was present.
- The *event name* of the event member is the name value of the <IDENTIFIER>.
- Each <event-declaration> within a class-module-declaration-section MUST specify a different event name.
- An event name can have the same name value as a module variable name, module constant name, enum member name, or procedure name that is defined within the same module.
- The name of an event MUST NOT contain any underscore characters (**Unicode** u+005F).
- *Runtime Semantics*
- Any <positional-param> elements contained in an <event-parameter-list> do not define any variables or variable bindings. They simply describe the arguments that MUST be provided to a <raiseevent-statement> that references the associated event name.

5.3 Module Code Section Structure

```
procedural-module-code-section = *( LINE-START  procedural-module-code-element LINE-END)
class-module-code-section = *( LINE-START  class-module-code-element LINE-END)

procedural-module-code-element = common-module-code-element
class-module-code-element = common-module-code-element / implements-directive

common-module-code-element = rem-statement / procedure-declaration

procedure-declaration = subroutine-declaration / function-declaration /       property-get-
declaration / property-LHS-declaration
```

There are several syntactic forms used to define procedures within the VBA Language. In some contexts of this specification it is necessary to refer to various kinds of declarations. The following table defines the kinds of declarations used in this specification and which grammar productions. If a checkmark appears in a cell, the kind of declaration defined in that column can refer to a declaration defined by that row's grammar production.

Grammar Rule	Procedure Declaration	Method Declaration	Property Declaration	Subroutine Declaration	Function Declaration
<subroutine-declaration>	√	√		√	
<function-declaration>	√	√			√
<external-sub>	√			√	
<external-	√				√

Grammar Rule	Procedure Declaration	Method Declaration	Property Declaration	Subroutine Declaration	Function Declaration
function>					
<property-get-declaration>	√	√	√		√
<property-lhs-declaration>	√	√	√	√	

5.3.1 Procedure Declarations

```
subroutine-declaration = procedure-scope [initial-static]
               "sub" subroutine-name [procedure-parameters] [trailing-static] EOS
                        [procedure-body EOS]
               [end-label] "end" "sub" procedure-tail

function-declaration = procedure-scope [initial-static]
                        "function" function-name [procedure-parameters] [function-type]
[trailing-static] EOS
                 [procedure-body EOS]
                  [end-label]  "end" "function" procedure-tail

property-get-declaration = procedure-scope [initial-static]
                 "Property" "Get"
                 function-name [procedure-parameters] [function-type] [trailing-static] EOS
                           [procedure-body EOS]
                           [end-label] "end" "property" procedure-tail

property-lhs-declaration = procedure-scope [initial-static]
                 "Property" ("Let" / "Set")
                  subroutine-name property-parameters [trailing-static] EOS
              [procedure-body EOS]
                 [end-label] "end" "property" procedure-tail

end-label = statement-label-definition
procedure-tail = [WS] LINE-END / single-quote comment-body /  ":" rem-statement
```

Static Semantics

- A function declaration implicitly defines a local variable, known as the function result variable, whose name and declared type are shared with the function and whose scope is the body of the function.
- A *function declaration* defines a procedure whose name is the name value of its <function-name> and a *subroutine declaration* defines a procedure whose name is the name value of its <subroutine-name>
- If the <function-name> element of a function declaration is a <TYPED-NAME> then the function declaration might not include a <function-type> element.
- The declared type of a function declaration is defined as follows:
  - If the <function-name> element of a function declaration is a <TYPED-NAME> then the declared type of the function declaration is the declared type corresponding to the <type-suffix> of the <TYPED-NAME> as specified in section 3.3.5.3.

- If the <function-name> element of a function declaration is not a <TYPED-NAME> and the function declaration does not include a <function-type> element its declared type is its implicit type as specified in section 5.2.3.1.5.
- If a function declaration includes a <function-type> element then the declared type of the function declaration is the specified type of the <function-type> element.

- The declared type of a function declaration that is part of a <class-module-code-section> might not be an UDT that is defined by a <private-type-declaration>.
- The declared type of a function declaration might not be a private enum name.
- If the optional <end-label> is present, its <statement-label> MUST have a label value that is different from the label value of any <statement-label> defined within the <procedure-body>.

Runtime Semantics

- The code contained by a procedure is executed during procedure invocation.
- Each invocation of a procedure has a distinct variable corresponding to each ByVal parameter or procedure extent variable declaration within the procedure.
- Each invocation of a function declaration has a distinct function result variable.
- A function result variable has procedure extent.
- Within the <procedure-body> of a procedure declaration that is defined within a <class-module-code-section> the declared type of the reserved name **Me** is the named class defined by the enclosing class module and the data value of "me" is an object reference to the object that is the target object of the currently active invocation of the function.
- Procedure invocation consists of the following steps:
  1. Create procedure extent variables corresponding to ByVal parameters.
  2. Process actual invocation augments as defined in section 5.3.1.11.
  3. Set the procedure's error handling policy (section *5.4.4*) to the default policy.
  4. Create the function result variable and any procedure extent local variables declared within the procedure.
  5. Execute the <procedure-body>.
  6. If the procedure is a function, return the data value of the result variable to the invocation site as the function result.
  7. The invocation is complete and execution continues at the call site.

5.3.1.1 Procedure Scope

```
procedure-scope = ["global" / "public" / "private" / "friend"]
```

Static Semantics

- A <procedure-declaration> that does not contain a <procedure-scope> element has the same meaning as if it included <procedure-scope> element consisting of the **Public** *keyword (section 3.3.5.1)*.

- A <procedure-declaration> that includes a <procedure-scope> element consisting of the **Public** *keyword* or **Global** *keyword* declares a *public procedure*. The procedure is accessible within the enclosing project. If the enclosing module is a class module or is a procedural module that is not a *private module*, then the procedure is also accessible within projects that reference the enclosing project.

- A <procedure-declaration> that includes a <procedure-scope> element consisting of the **Friend** *keyword* declares a *friend procedure*. The procedure is accessible within the enclosing project.

- A <procedure-declaration> that includes a <procedure-scope> element consisting of the **Private** *keyword* declares a *private procedure*. The procedure is accessible within the enclosing module.

- A <procedure-scope> consisting of the *keyword* **Global** might not be an element of a <procedure-declaration> contained in a <class-module-code-section>

- A <procedure-scope> consisting of the *keyword* **Friend** might not be an element of a <procedure-declaration> contained in a <procedural-module-code-section>

5.3.1.2 Static Procedures

```
initial-static = "static"
trailing-static = "static"
```

Static Semantics

- A <procedure-declaration> containing either an <initial-static> element or a <trailing-static> element declares a *static procedure*.

- No <procedure-declaration> contains both an <initial-static> element and a <trailing-static> element.

Runtime Semantics

- All variables declared within the <procedure-body> of a static procedure have module extent.

- All variables declared within the <procedure-body> of a non-static procedure have procedure extent.

5.3.1.3 Procedure Names

```
subroutine-name = IDENTIFIER / prefixed-name
function-name = TYPED-NAME / subroutine-name
prefixed-name = event-handler-name / implemented-name / lifecycle-handler-name
```

Static Semantics

- The *procedure name* of a procedure declaration is the name value of its contained <subroutine-name> or <function-name> element.

- If a procedure declaration whose visibility is public has a procedure name that is the same as the name of a project or name of a module then all references to the procedure name MUST be

explicitly qualified with its project or module name unless the reference occurs within the module that defines the procedure.

5.3.1.4 Function Type Declarations

```
function-type = "as" type-expression [array-designator]
array-designator = "(" ")"
```

Static Semantics

- The specified type of a <function-type> that does not include an <array-designator> element is the declared type referenced by its <type-expression> element.
- The specified type of a <function-type> that includes an <array-designator> element is resizable array with a declared element type that is the declared type referenced by its <type-expression> element.

5.3.1.5 Parameter Lists

```
procedure-parameters = "(" [parameter-list] ")"
property-parameters = "(" [parameter-list ","] value-param ")"

parameter-list = (positional-parameters "," optional-parameters ) /
                 (positional-parameters  ["," param-array]) /
                 optional-parameters /
                 param-array

positional-parameters = positional-param *("," positional-param)
optional-parameters = optional-param *("," optional-param)
value-param = positional-param
positional-param = [parameter-mechanism] param-dcl
optional-param = optional-prefix param-dcl [default-value]
param-array = "paramarray" IDENTIFIER "(" ")" ["as" ("variant" / "[variant]")]

param-dcl = untyped-name-param-dcl / typed-name-param-dcl
untyped-name-param-dcl = IDENTIFIER [parameter-type]
typed-name-param-dcl = TYPED-NAME [array-designator]
optional-prefix = ("optional" [parameter-mechanism]) / ([parameter-mechanism] ("optional"))
parameter-mechanism = "byval" / " byref"
parameter-type = [array-designator] "as" (type-expression / "Any")
default-value = "=" constant-expression
```

Static Semantics

- A <parameter-type> element only include the *keyword* **Any** if the <parameter-type> is part of a <external-proc-dcl>.
- The name value of a <typed-name-param-dcl> is the name value of its <TYPED-NAME> element.
- The name value of an <untyped-name-param-dcl> is the name value of its <IDENTIFIER> element.
- The name value of a <param-dcl> is the name value of its constituent <untyped-name-param-dcl> or <typed-name-param-dcl> element.
- The name of a <positional-param> or a <optional-param> element is the name value of its <param-dcl> element.
- The name of a <param-array> element is the name value of its <IDENTIFIER> element.

- Each <positional-param>, <optional-param>, and <param-array> that are elements of the same <parameter-list>, <property-parameters>, or <event-parameter-list> MUST have a distinct names.
- The name of each <positional-param>, <optional-param>, and <param-array> that are elements of a function declaration MUST be different from the name of the function declaration.
- The name value of a <positional-param>, <optional-param>, or a <param-array> might not be the same as the name of any variable defined by a <local-variable-declaration>, a <static-variable-declaration>, a <redim-statement>, or a <local-const-declaration> within the <procedure-body> of the containing procedure declaration.
- The declared type of a <positional-param>, <optional-param>, or <value-param> is the declared type of its constituent <param-dcl>.
- The declared type of a <param-dcl>that consists of an <untyped-name-param-dcl>is defined as follows:
  - If the optional <parameter-type> element is not present, the declared type is the implicit type of the <IDENTIFIER> as specified in section 5.2.3.1.5.
  - If the specified optional <parameter-type> element is present but does not include an <array-designator> element the declared type is the declared type referenced by its <type-expression> element.
  - If the specified optional <parameter-type> element is present and includes an <array-designator> element the declared type is resizable array whose element type is the declared type referenced by its <type-expression> element.
- The declared type of a <param-dcl> that consists of a <typed-name-param-dcl> is defined as follows:
  - If the optional <array-designator> element is not present the declared type is the declared type corresponding to the <type-suffix> of the <TYPED-NAME> as specified in section 3.3.5.3.
  - If the optional <array-designator> element is present then the declared type of the defined variable is resizable array with a declared element type corresponding to the <type-suffix> of the <TYPED-NAME> as specified in section 3.3.5.3.
- The declared type of a <param-dcl> that is contained in an event declaration or a public procedure declaration in a <class-module-code-section> might not be a private UDT, a public UDT defined in a procedural module, or a private enum name.
- The declared type of an <optional-param> might not be an UDT.
- If the declared type of an <optional-param> is not Variant and its type was implicitly specified by an applicable <def-directive>, it MUST have a <default-value> clause specified.
- A <default-value> clause specifies the default value of a parameter. If a <default-value> clause is not specified for a **Variant** parameter, the default value is an implementation-defined error value that resolves to standard error code 448 (“Named argument not found”). If a <default-value> clause is not specified for a non-**Variant** parameter, the default value is that of the parameter’s declared type.
- A <positional-param> or <optional-param> element that does not include a <parameter-mechanism> element has the same meaning as if it included a <parameter-mechanism> element consisting of the *keyword* **ByRef**.
- A <param-dcl> that includes a <parameter-mechanism> element consisting of the *keyword* **ByVal** might not also include an <array-designator> element.

- The declared type of the <IDENTIFIER> of a <param-array>is resizable array of Variant.

  Runtime Semantics

- Each invocation of a function has a distinct function result variable.
- A function result variable has procedure extent.
- Each <positional-param> or <optional-param> that includes a <parameter-mechanism> element consisting of the *keyword* **ByVal** defines a local variable with procedure extent and whose declared type is the declared type of the constituent <param-dcl> element. The corresponding parameter name is bound to the local variable.
- Each <positional-param> that includes a <parameter-mechanism> element consisting of the *keyword* **ByVal** defines a local name binding to a pre-existing variable corresponding to the corresponding positional argument.
- Each <optional-param> that includes a <parameter-mechanism> element consisting of the *keyword* **ByRef** defines a local variable with procedure extent and whose declared type is the declared type of the constituent <param-dcl> element.
  - If an invocation of the containing procedure does not include an argument corresponding to the <optional-param> the parameter name is bound to the local variable for that invocation.
  - If an invocation of the containing procedure includes an argument corresponding to the <optional-param> the parameter name is locally bound to the pre-existing variable corresponding to the argument.
- Upon invocation of a procedure the data value of the constituent <default-value> element of each <optional-param> that does not have a corresponding argument is assigned to the variable binding of the parameter name of the <optional-param>.
- Each procedure that is a method has an implicit ByVal parameter called the *current object* that corresponds to the target object of an invocation of the method. The current object acts as an anonymous local variable with procedure extent and whose declared type is the class name of the class module containing the method declaration. For the duration of an activation of the method the data value of the current object variable is target object of the procedure invocation that created that activation. The current object is accessed using the **Me** keyword within the <procedure-body> of the method but cannot be assigned to or otherwise modified.
- If a <parameter-list> of a procedure contains a <param-array> element, then each invocation of the procedure defines an entity called the *param array* that behaves as if it was an array whose elements were "byref" <positional-param> elements whose declared types were **Variant**. An access to an element of the param array behaves as if it were an access to a named positional parameter. Arguments are bound to the elements of a param array as defined in section 5.3.1.11.

5.3.1.6 Subroutine and Function Declarations

Static Semantics

- Each <subroutine-declaration> and <function-declaration> MUST have a procedure name that is different from any other module variable name, module constant name, enum member name, or procedure name that is defined within the same module.

5.3.1.7 Property Declarations

Static Semantics

- A <property-LHS-declaration> containing the keyword **Let** is a *property let declaration*.
- A <property-LHS-declaration> containing the keyword **Set** is a *property set declaration*.
- Each property declaration MUST have a procedure name that is different from the name of any other module variable, module constant, enum member name, external procedure, <function-declaration>, or <subroutine-declaration> that is defined within the same module.
- Each <property-get-declaration> in a module MUST have a different name.
- Each property let declaration in a module MUST have a different name.
- Each property set declaration in a module MUST have a different name.
- Within a module at a common procedure name can be shared by a <property-get-declaration>, a property let declaration, and a property set declaration.
- Within a module all property declaration that share a common procedure name MUST have equivalent <parameter-list> elements including the number of <positional-parameters>, <optional-parameters> and <param-array> elements, the name value of each corresponding parameter, the declared type of each corresponding parameter, and the actual <parameter-mechanism> used for each corresponding parameter. However, corresponding <optional-param> elements can differ in the presence and data value of their <default-value> elements and as can whether or not the <parameter-mechanism> is implicitly specified or explicitly specified.
- The declared type of a <property-LHS-declaration> is the declared type of its <value-param> element.
- The declared type of a property set declaration MUST be **Object**, **Variant**, or a named class.
- Within a module a property let declaration and a <property-get-declaration> that share a common procedure name MUST have the same declared type.
- If the <value-param> of a <property-LHS-declaration> does not have a <parameter-mechanism> element or has a <parameter-mechanism> consisting of the *keyword* **ByRef**, it has the same meaning as if it instead had a <parameter-mechanism> element consisting of the *keyword* **ByVal**.

Runtime Semantics

- The <value-param> of a <property-LHS-declaration> always has the runtime semantics of a ByVal parameter.
- If a <property-LHS-declaration> includes a <param-array> element the argument value corresponding to the <value-param> in an invocation of the property is not included as an element of its param array.

5.3.1.8 Event Handler Declarations

```
event-handler-name = IDENTIFIER
```

Static Semantics

- A procedure declaration qualifies as an *event handler* if all of the following are true:
  - It is contained within a class module.

- The name value of the subroutine name MUST begin with an event handler name prefix corresponding to a *WithEvents variable* declaration within the same class module as the procedure declaration. The variable defined by the corresponding variable declaring declaration is called the *associated variable* of the event handler.
- The procedure name text that follows the event handler name prefix MUST be the same as an event name defined by the class that is the declared type of the associated variable. The corresponding <event-declaration> is the handled event.

- An event handler is invalid if any of the following are true:
  - The procedure declaration is not a <subroutine-declaration>.
  - Its <parameter-list> is not compatible with the <event-parameter-list> of the handled event. A compatible <parameter-list> is one that meets all of the following criteria:
    - The number of <positional-parameters> elements MUST be the same.
    - Each corresponding parameter has the same type and parameter mechanism. However, corresponding parameters can differ in name and in whether the <parameter-mechanism> is specified implicitly or explicitly.

5.3.1.9 Implemented Name Declarations

```
implemented-name = IDENTIFIER
```

Static Semantics

- A procedure declaration qualifies as an *implemented name declaration* if all of the following are true:
  - The name value of the procedure name MUST begin with an implemented interface name prefix defined by an <implements-directive> within the same class module. The class identified by <class-type-name> element of the corresponding <implements-directive> is called the interface class.
  - The procedure name text that follows the implemented interface name prefix MUST be the same as the name of a corresponding public variable or method defined by the interface class. The corresponding variable or method is called the interface member.
  - If the interface member is a variable declaration then the candidate implemented method declaration MUST be a property declaration.
  - If the interface member is a method declaration then the candidate implemented method MUST be the same kind (<function-declaration>, <subroutine-declaration>, <property-get-declaration>, <property-lhs-declaration>) of method declaration.
- An implemented name declaration whose corresponding interface member is a method MUST have an <procedure-parameters> or <property-parameters> element that is equivalent to the <procedure-parameters> or <property-parameters> element of the interface member according to the following rules:
  - The <parameter-list> elements including the number of <positional-parameters>, <optional-parameters> and <param-array> elements, the declared type of each corresponding parameter, the constant values of the <default-value> of corresponding <optional-parameters> elements, and the actual <parameter-mechanism> used for each corresponding parameter. However, corresponding <parameter-list> elements can differ in their parameter

names and whether or not the <parameter-mechanism> is implicitly specified or explicitly specified.

  - If the corresponding members are property set declarations or property get declarations their <value-param> elements MUST be equivalent according to the preceding rule.

- If the interface member is a function declaration then the declared type of the function defined by the implemented name declaration and the declared type of the function defined by the interface member but be the same.

- If the interface member is a variable and the implemented name declaration is a property declaration the declared type of the implemented name property declaration MUST be the same as the declared type of the interface member.

Runtime Semantics

- When the target object of an invocation has a declared type that is an interface class of the actual target object's class and the method name is the name of an interface member of that interface class then the actual invoked method is the method defined by the corresponding implemented method declaration of target's object's class.

5.3.1.10 Lifecycle Handler Declarations

```
lifecycle-handler-name = "Class_Initialize" / "Class_Terminate"
```

Static Semantics

- A lifecycle handler declaration is a subroutine declaration that meets all of the following criteria:

  - It is contained within a class module.

  - It's procedure name is a <lifecycle-handler-name>

  - The <procedure-parameters> element of the <subroutine-declaration> is either not present or does not contain a <parameter-list> element

Runtime Semantics

- If a class defines a *Class_Initialize lifecycle handler*, that subroutine will be invoked as an method each time an instance of that class is created by the **New** operator, by referencing a variable that was declared with an <as-auto-object> and whose current value is **Nothing**, or by call the CreateObject function (section 6.1.2.8.1.4) of the VBA Standard Library. The target object of the invocation is the newly created object. The invocation occurs before a reference to the newly created object is returned from the operations that creates it.

- If a class defines a *Class_Terminate lifecycle handler*, that subroutine will be invoked as an method each time an instance of that class is about to be destroyed. The target object of the invocation is the object that is about to be destroyed. The invocation of a *Class_Terminate lifecycle handler* for an object can occur at precisely at the point the object becomes provably inaccessible to VBA program code but can occur at some latter point during execution of the program

- In some circumstances, a *Class_Terminate lifecycle handler* can cause the object to cease to be provably inaccessible. In such circumstances, the object is not destroyed and is no longer a candidate for destruction. However, if such an object later again becomes provably inaccessible it can be destroyed but the *Class_Terminate lifecycle handler* will not be invoked again for that

target object. In other words, a "Class_Terminate" lifecycle handler executes at most once during the lifetime of an object.

- If the error-handling policy of a *Class_Terminate lifecycle handler* is to use the error-handling policy of the procedure that invoked it, the effect is as if the *Class_Terminate lifecycle handler* was using the default error-handling policy. This means that errors raised in a *Class_Terminate lifecycle handler* can only be handled in the handler itself.

5.3.1.11 Procedure Invocation Argument Processing

A *procedure invocation* consists of a procedure expression, classified as a property, function or subroutine, an argument list consisting of positional and/or named arguments, and, if the procedure is defined in a class module, a target object.

Static semantics.

The argument expressions contained within the *argument list* at the site of invocation are considered the *arguments*. When the procedure expression is classified as a property, function or subroutine, the argument list is statically checked for *compatibility* with the *parameters* defined in the declaration of the referenced procedure as follows:

- The arguments are first mapped to the parameters as follows:
  - Each *positional argument* specified is mapped in order from left to right to its respective positional parameter. If there are more positional arguments than there are parameters, the argument list is incompatible, unless the last parameter is a param array. If a positional argument is specified with its value omitted and its mapped parameter is not optional, the argument list is incompatible, even if a named argument is later mapped to this parameter.
  - Each *named argument* is mapped to the parameter with the same name value. If there is no parameter with the same name value, or if two or more named or positional arguments are mapped to the same parameter, the argument list is incompatible.
- If any non-optional parameter does not have an argument mapped to it, the argument list is incompatible.
- For each mapped parameter:
  - If the parameter is *ByVal*:
    - If the parameter has a declared type other than a specific class or **Object**, and a **Let**-coercion from the declared type of its mapped argument to the parameter's declared type is invalid, the argument list is incompatible.
    - If the parameter has a declared type of a specific class or **Object**, and the declared type of its mapped argument is a type other than a specific class, **Object**, or **Variant**, the argument list is incompatible.
  - Otherwise, if the parameter is *ByRef*:
    - If the parameter has a declared type other than a specific class, **Object** or **Variant**, and the declared type of the parameter does not exactly match that of its mapped argument, the argument list is incompatible.
    - If the parameter has a declared type of a specific class or **Object**, and the declared type of its mapped argument is a type other than a specific class or **Object**, the argument list is incompatible.

A procedure invocation is invalid if the argument list is statically incompatible with the parameter list.

Runtime semantics.

The runtime semantics of procedure invocation for procedures are as follows:

- The arguments are first mapped to the parameters as follows:
  - Each positional argument specified is mapped in order from left to right to its respective positional parameter. If there are more positional arguments than there are parameters, runtime error 450 (Wrong number of arguments or invalid property assignment) is raised, unless the last parameter is a param array, in which case the param array is set to a new array of element type **Variant** with a lower bound of 0 containing the extra arguments in order from left to right. If a positional argument is specified with its value omitted and its mapped parameter is not optional, runtime error 448 (Named argument not found) is raised, even if a named argument is later mapped to this parameter.
  - Each named argument is mapped to the parameter with the same name value. If there is no parameter with the same name value, or if two or more named or positional arguments are mapped to the same parameter, runtime error 448 (Named argument not found) is raised.
  - If the last parameter is a param array and there are not more positional arguments than there are parameters, the param array is set to a new array of element type **Variant** with a lower bound of 0 and an upper bound of -1.
- If any non-optional parameters does not have an argument mapped to it, runtime error 449 (Argument not optional) is raised.
- For each parameter, in order from left to right:
  - If the parameter has no argument mapped to it, the parameter is *ByVal*, or the parameter is *ByRef* and the mapped argument's expression is classified as a value, function, property or unbound member, a local variable is defined with procedure extent within the procedure being invoked with the same name value and declared type as the parameter, and has its value assigned as follows:
    - If this parameter is optional and has no argument mapped to it, the parameter's default value is assigned to the new local variable.
    - If the value type of this parameter's mapped argument is a type other than a specific class or **Nothing**, the argument's data value is **Let**-assigned to the new local variable.
    - Otherwise, if the value type of this parameter's mapped argument is a specific class or **Nothing**, the argument's data value is **Set**-assigned to the new local variable.
  - Otherwise, if the parameter is *ByRef* and the mapped argument's expression is classified as a variable:
    - If the declared type of the parameter is a type other than a specific class, **Object** or **Variant**, a reference parameter binding is defined within the procedure being invoked, with the same name and declared type as the parameter, referring to the variable referenced by the argument's expression.
    - If the declared type of the parameter is a specific class or **Object**:
      - If the declared type of the formal exactly matches the declared type of the argument's expression, a reference parameter binding is defined within the procedure being invoked, with the same name and declared type as the parameter, referring to the variable referenced by the argument's expression.
      - If the declared type of the formal does not exactly match the declared type of the argument's expression:

        - A local variable is defined with procedure extent within the procedure being invoked with the same name value and declared type as the parameter, with the argument's value **Set**-assigned to the new local variable.
        - When the procedure terminates, if it has terminated normally, the value within the local variable is **Set**-assigned back to the argument's referenced variable.
    - If the declared type of the parameter is **Variant**, a reference parameter binding is defined within the procedure being invoked, with the same name as the parameter, referring to the variable referenced by the argument's expression. This reference parameter binding is treated as having a declared type of **Variant**, except when used as the <l-expression> within **Let**-assignment or **Set**-assignment, in which case it is treated as having the declared type of the argument's referenced variable.
- For each unmapped optional parameter, a local variable is defined with procedure extent within the procedure being invoked with the same name value and declared type as the parameter, and has its value assigned as follows:
    - If the parameter has a specified default value other than **Nothing**, this default value is **Let**-assigned to the new local variable.
    - If the parameter has a specified default value of Nothing, this default value is **Set**-assigned to the new local variable.
    - If the parameter has no specified default value, the new local variables is initialized to the default value for its declared type.

There can be implementation-specific differences in the semantics of parameter passing during invocation of procedures imported from a library project.

5.4 Procedure Bodies and Statements

Procedure bodies contain the imperative statements that describe the algorithmic actions of a VBA procedure. A procedure body also includes definitions of statement labels and declarations for local variables whose usage is private to the procedure.

```
procedure-body = statement-block
```

Static Semantics

- The label values of all <statement-label-definition> elements within the <statement-block> and any lexically contained <statement-block> elements MUST be unique.
- The label values of all <statement-label-definition> elements within the <statement-block> of a <procedure-body> MUST be distinct from the label value of the <end-label> of the containing procedure declaration.

5.4.1 Statement Blocks

A *statement block* is a sequence of 0 or more statements.

```
statement-block = *(block-statement EOS)

block-statement = statement-label-definition / rem-statement / statement

statement = control-statement / data-manipulation-statement / error-handling-statement /
file-statement
```

Runtime Semantics

- Execution of a <statement-block> starts by executing the first <block-statement> contained in the block and continues in sequential order until either the last contained <block-statement> is executed or a <control-statement> explicitly transfers execution to a <statement-label-definition> that is not contained in the <statement-block>.
- Execution of a <statement-block> can begin by a <control-statement> transferring execution to a <statement-label-definition> contained within the <statement-block>. In that case, execution sequential statement execution begins with the target <statement-label-definition> and any <block-statement> elements preceding the target <statement-label-definition> are not executed.
- <control-statement> elements within a <statement-block> can modify sequential execution order by transferring the current point of execution to a <statement-label-definition> contained within the same <statement-block>.
- An identifier followed by ":" at the beginning of a line is always interpreted as a <statement-label-definition> rather than a <statement>.

5.4.1.1 Statement Labels

```
statement-label-definition = LINE-START ((identifier-statement-label ":") / (line-number-
label [":"] ))
statement-label = identifier-statement-label / line-number-label
statement-label-list = statement-label ["," statement-label]
identifier-statement-label = IDENTIFIER
line-number-label = INTEGER
```

Static Semantics.

- The name value of the <IDENTIFIER> in <identifier-statement-label> might not be "Randomize". ☐ If <statement-label> is an <INTEGER>, it data value MUST be in the inclusive range 0 to 2,147,483,647.
- The *label value* of a <statement-label-definition> is the label value of its constituent <identifier-statement-label> or its constituent <line-number-label>.
- The label value of a <statement-label> is the label value of its constituent <identifier-statement-label> or its constituent <line-number-label>.
- The label value of an <identifier-statement-label> is the name value of its constituent <IDENTIFIER> element.
- The label value of a <line-number-label> is the data value of its constituent <INTEGER> element.
- It is an error for a procedure declaration to contain more than one <statement-label-definition> with the same label value.

Runtime Semantics.

- Executing a <statement-label-definition> has no observable effect.

5.4.1.2 Rem Statement

A <rem-statement> contains program commentary text that is that has no effect upon the meaning of the program.

```
rem-statement = "Rem" comment-body
```

Runtime Semantics.

- Executing a <rem-statement> has no observable effect.

5.4.2 Control Statements

Control statements determine the flow of execution within a program.

```
control-statement = if-statement / control-statement-except-multiline-if

control-statement-except-multiline-if = call-statement / while-statement / for-statement /
exit-for-statement / do-statement / exit-do-statement / single-line-if-statement /  select-
case-statement /stop-statement / goto-statement / on-goto-statement / gosub-statement /
return-statement / on-gosub-statement /for-each-statement / exit-sub-statement / exit-
function-statement / exit-property-statement / raiseevent-statement / with-statement
```

5.4.2.1 Call Statement

A <call-statement> invokes a subroutine or function, discarding any return value.

```
call-statement = "Call" (simple-name-expression / member-access-expression / index-expression
/ with-expression)
call-statement =/ (simple-name-expression / member-access-expression / with-expression)
argument-list
```

Static semantics.

- If the **Call** keyword is omitted, the first positional argument, if any, can only represent a <with-expression> if it is directly preceded by whitespace.
- The specified argument list is determined as follows:
  - If the **Call** keyword is specified:
    - If a <call-statement> element's referenced expression is an <index-expression>, the *specified argument list* is this expression's argument list.
    - Otherwise, the specified argument list is an empty argument list.
  - Otherwise, if the **Call** keyword is omitted, the specified argument list is <argument-list>.
- A **<**call-statement> is invalid if any of the following is true:
  - The referenced expression is not classified as a variable, function, subroutine or unbound member.
  - The referenced expression is classified as a variable and one of the following is true:
    - The declared type of the referenced expression is a type other than a specific class or **Object**.
    - The declared type of the referenced expression is a specific class without a default function or subroutine.

        - The declared type of the referenced expression is a specific class with a default function or subroutine whose parameter list is incompatible with the specified argument list.

- The referenced expression is classified as a function or subroutine and its referenced procedure's parameter list is incompatible with the specified argument list.

Runtime semantics.

At runtime, the procedure referenced by the expression is invoked, as follows:

- If the expression is classified as an unbound member, the member is resolved as a variable, property, function or subroutine, and evaluation continues as if the expression had statically been resolved as a variable expression, property expression, function expression or subroutine expression, respectively.
- If the expression is classified as a function or subroutine, the expression's referenced procedure is invoked with the specified argument list. Any return value resulting from the invocation is discarded.
- If the expression is classified as a variable:
    - If the expression's data value is an object with a public default function or subroutine, this default procedure is invoked with the specified argument list.
    - If the expression's data value is an object with a public default property, runtime error 450 (Wrong number of arguments or invalid property assignment) is raised.
    - Otherwise, runtime error 438 (Object doesn't support this property or method) is raised.
- If the expression is classified as a property, runtime error 450 (Wrong number of arguments or invalid property assignment) is raised.

5.4.2.2 While Statement

A <while-statement> executes a sequence of statements as long as a specified pre-condition is True.

```
while-statement = "While" boolean-expression EOS  statement-block  "Wend"
```

Runtime Semantics.

The <boolean-expression> is repeatedly evaluated until the value of an evaluation is the data value False. Each time an evaluation of the <boolean-expression> has the data value True, the <statement-block> is executed prior to the next evaluation of <boolean-expression>.

5.4.2.3 For Statement

A <for-statement> executes a sequence of statements a specified number of times.

```
for-statement = simple-for-statement / explicit-for-statement

simple-for-statement = for-clause EOS statement-block "Next"

explicit-for-statement = for-clause EOS statement-block
("Next" / (nested-for-statement ",")) bound-variable-expression
nested-for-statement = explicit-for-statement / explicit-for-each-statement
for-clause = "For" bound-variable-expression "=" start-value "To" end-value [step-clause]
start-value = expression
end-value = expression
```

```
step-clause = Step" step-increment
step-increment = expression
```

Static Semantics.

- If no <step-clause> is present, the <step-increment> value is the integer data value 1.
- The <bound-variable-expression> within the <for-clause> of an <explicit-for-statement> MUST resolve to the same variable as the <bound-variable-expression> following the <statement-block>. The declared type of <bound-variable-expression> MUST be a numeric value type or **Variant**.
- The declared type of <start-value>, <end-value>, and <step-increment> MUST be statically **Let**-coercible to **Double**.

Runtime Semantics.

- The expressions <start-value>, <end-value>, and <step-increment> are evaluated once, in order, and prior to any of the following computations. If the value of <start-value>, <end-value>, and <step-increment> are not Let-coercible to **Double**, error 13 (Type mismatch) is raised immediately. Otherwise, proceed with the following algorithm using the original, uncoerced values.
- Execution of the <for-statement> proceeds according to the following algorithm:
  1. If the data value of <step-increment> is zero or a positive number, and the value of <bound-variable-expression> is greater than the value of <end-value>, then execution of the <for-statement> immediately completes; otherwise, advance to Step 2.
  2. If the data value of <step-increment> is a negative number, and the value of <bound-variable-expression> is less than the value of <end-value>, execution of the <for-statement> immediately completes; otherwise, advance to Step 3.
  3. The <statement-block> is executed. If a <nested-for-statement> is present, it is then executed. Finally, the value of <bound-variable-expression> is added to the value of <step-increment> and Let-assigned back to <bound-variable-expression>. Execution then repeats at step 1.
- If a <goto-statement> defined outside the <for-statement> causes a <statement> within <statement-block> to be executed, the expressions <start-value>, <end-value>, and <step-increment> are not evaluated. If execution of the <statement-block> completes and reaches the end of the <statement-block> without having evaluated <start-value>, <end-value> and <step-increment> during this execution of the enclosing procedure, an error is generated (number 92, "For loop not initialized"). This occurs even if <statement-block> contains an assignment expression that initializes <bound-variable-expression> explicitly. Otherwise, if the expressions <start-value>, <end-value>, and <step-increment> have already been evaluated, the algorithm continues at Step 3 according to the rules defined for execution of a <for-statement>.
- When the <for-statement> has finished executing, the value of <bound-variable-expression> remains at the value it held as of the loop completion.

5.4.2.4 For Each Statement

A <for-each-statement> executes a sequence of statements once for each element of a collection.

```
for-each-statement = simple-for-each-statement / explicit-for-each-statement

simple-for-each-statement = for-each-clause EOS statement-block "Next"
```

```
explicit-for-each-statement = for-each-clause EOS statement-block
 ("Next" / (nested-for-statement ",")) bound-variable-expression

for-each-clause = "For" "Each" bound-variable-expression "In" collection
collection = expression
```

Static Semantics.

- The <bound-variable-expression> within the <for-each-clause> of an <explicit-for-each-statement> MUST resolve to the same variable as the <bound-variable-expression> following the keyword **Next**.
- If the declared type of <collection> is array then the declared type of <bound-variable-expression> MUST be **Variant**.

Runtime Semantics.

- The expression <collection> is evaluated once prior to any of the following computations.
- If the data value of <collection> is an array:
  - If the array has no elements, then execution of the <for-each-statement> immediately completes.
  - If the declared type of the array is **Object**, then the <bound-variable-expression> is **Set**-assigned to the first element in the array. Otherwise, the <bound-variable-expression> is Let-assigned to the first element in the array.
  - After <bound-variable-expression> has been set, the <statement-block> is executed. If a <nested-for-statement> is present, it is then executed.
  - Once the <statement-block> and, if present, the <nested-for-statement> have completed execution, <bound-variable-expression> is Let-assigned to the next element in the array (or Set-assigned if it is an array of Object). If and only if there are no more elements in the array, then execution of the <for-each-statement> immediately completes. Otherwise, <statement-block> is executed again, followed by <nested-for-statement> if present, and this step is repeated.
  - When the <for-each-statement> has finished executing, the value of <bound-variable-expression> is the data value of the last element of the array.
- If the data value of <collection> is not an array:
  - The data value of <collection> MUST be an object-reference to an external object that supports an implementation-defined enumeration interface. The <bound-variable-expression> is either Let-assigned or Set-assigned to the first element in <collection> in an implementation-defined manner.
  - After <bound-variable-expression> has been set, the <statement-block> is executed. If a <nested-for-statement> is present, it is then executed.
  - Once the <statement-block> and, if present, the <nested-for-statement> have completed execution, <bound-variable-expression> is Set-assigned to the next element in <collection> in an implementation-defined manner. If there are no more elements in <collection>, then execution of the <for-each-statement> immediately completes. Otherwise, <statement-block> is executed again, followed by <nested-for-statement> if present, and this step is repeated.

- When the <for-each-statement> has finished executing, the value of <bound-variable-expression> is the data value of the last element in <collection>.

- If a <goto-statement> defined outside the <for-each-statement> causes a <statement> within <statement-block> to be executed, the expression <collection> is not evaluated. If execution of the <statement-block> completes and reaches the end of the <statement-block> without having evaluated <collection> during this execution of the enclosing procedure, an error is generated (number 92, "For loop not initialized"). This occurs even if <statement-block> contains an assignment expression that initializes <bound-variable-expression> explicitly. Otherwise, if the expression <collection> has already been evaluated, the algorithm continues according to the rules defined for execution of a <for-each-statement> over the <collection>.

5.4.2.4.1 Array Enumeration Order

- When enumerating the elements of an array, the *first element* is defined to be the element at which all array indices are at the lower bound of their respective array dimensions.

- The *next element* is the obtained by incrementing the array index at the leftmost dimension. If incrementing a dimension brings it above its upper bound, that dimension is set to its lower bound and the next dimension to the right is incremented.

- The *last element* is defined to be the element at which all array indices are at the upper bound of their respective array dimensions.

5.4.2.5 Exit For Statement

```
exit-for-statement = "Exit" "For"
```

Static Semantics.

- An <exit-for-statement> MUST be lexically contained inside a <for-statement> or a <for-each-statement>.

Runtime Semantics.

- Execution of the closest lexically-enclosing <for-statement> or <for-each-statement> enclosing this statement immediately completes. No other statements following the <exit-for-statement> in its containing <statement-block> are executed.

5.4.2.6 Do Statement

A <do-statement> executes a sequence of statements as long as a specified pre/post-condition is True.

```
do-statement = "Do" [condition-clause] EOS statement-block

               "Loop" [condition-clause]

condition-clause = while-clause / until-clause

while-clause = "While" boolean-expression
until-clause = "Until" boolean-expression
```

Static Semantics.

- Only one <condition-clause> can be specified after the keyword **Do** or the keyword **Loop**, not both. If an <until-clause> is specified, the effect is as if it were a <while-clause> with the value of the <boolean-expression> set to "Not (<boolean-expression>)".
- If no <condition-clause> is specified (either after **Do** or **Loop**), the effect is as if a <condition-clause> containing a <while-clause> with the expression "**True**" were specified after **Do**.

Runtime Semantics.

- A <do-statement> repeatedly evaluates its <condition-clause> and executes the <statement-block> if it evaluates to the data value **True**. The ordering of the of the evaluation of the <condition-clause> and the execution of the <statement-block> is defined by the following table:

Location of <condition-clause>	Result
None specified	Execution of the loop continues until an <exit-do-statement> is executed.
Immediately following "Do"	<condition-clause> is evaluated prior to executing <statement-block>. If it evaluates to the data value **False** then execution of the <statement-block> and the current statement immediately completes.
Immediately following "Loop"	The <statement-block> is executed before evaluation of the <condition-clause>. If it evaluates to the data value **True**, then the <statement-block> is again executed and the process is repeated. If it evaluates to the data value **False** then execution of the <statement-block> and the current statement immediately completes.

5.4.2.7 Exit Do Statement

exit-do-statement = "Exit" "Do"

Static Semantics.

- An <exit-do-statement> MUST be lexically contained inside a <do-statement>.

Runtime Semantics.

- If the <statement-block> causes execution of an <exit-do-statement> whose closest lexically containing <do-statement> is this statement, execution of the <statement-block> and of this statement immediately completes. No other statements following the <exit-do-statement> in the <statement-block> are executed.

5.4.2.8 If Statement

An <if-statement> determines whether or not to execute a <statement-block>.

```
if-statement = LINE-START "If" boolean-expression "Then" EOL statement-block
               *[else-if-block]
               [else-block]
               LINE-START (("End" "If") / "EndIf")
else-if-block = LINE-START "ElseIf" boolean-expression "Then" EOL
                LINE-START statement-block
else-if-block =/ "ElseIf" boolean-expression "Then" statement-block
else-block = LINE-START "Else" statement-block
```

Runtime Semantics.

- An <if-statement> evaluates its <boolean-expression>, and if it equals the data value **True**, it executes the <statement-block> after "Then". If it equals the data value False, execution continues in the following order:

  1. The <boolean-expression> in each <else-if-block> (in order) is evaluated, until a <boolean-expression> whose data value is **True** is encountered. The <statement-block> of the containing <else-if-block> is executed and completes execution of the <if-statement>

  2. If none of the <boolean-expression> in the <else-if-block>s equal the data value **True**, and an <else-block> is present, the <statement-block> of the <else-block> is executed.

- If a <goto-statement> defined outside the <if-statement> causes a <statement> within <statement-block> to be executed, the <boolean-expression> is not evaluated. A <goto-statement> can also cause execution to leave the <statement-block>. If a later <goto-statement> causes execution to re-enter the <statement-block>, the behavior is as specified by the rules defined for execution of an <if-statement>.

5.4.2.9 Single-line If Statement

A <single-line-if-statement> determines whether or not to execute a statement.

```
single-line-if-statement = if-with-non-empty-then / if-with-empty-then

if-with-non-empty-then = "If" boolean-expression "Then" list-or-label [single-line-else-
clause]
if-with-empty-then = "If" boolean-expression "Then" single-line-else-clause
single-line-else-clause = "Else" [list-or-label]
list-or-label = (statement-label *[":" [same-line-statement]]) /
([":"] same-line-statement *[":" [same-line-statement]])
same-line-statement = file-statement / error-handling-statement /
data-manipulation-statement / control-statement-except-multiline-if
```

Static Semantics.

- A <single-line-if-statement> is distinguished from an <if-statement> by the presence of a <list-or-label> or a <single-line-else-clause> immediately following the **Then** keyword.

- A <single-line-if-statement> MUST be defined on a single logical line, including the entirety of any occurrence of a <same-line-statement>. This restriction precludes any embedded <EOS> alternatives that require a <LINE-END> element.

- When the <list-or-label> of a <single-line-if-statement> contains a <single-line-if-statement>, the first <single-line-else-clause> is part of the immediately preceding <single-line-if-statement>. Any subsequent <single-line-else-clause>is paired with the first <single-line-if-statement> preceding the already paired if-then-else-statements.

- A <statement-label> that occurs as the first element of a <list-or-label> element has the effect as if the <statement-label> was replaced with a <goto-statement> containing the same <statement-label>. This <goto-statement> takes the place of <line-number-label> in <statement-label-list>.

Runtime Semantics.

- A <single-line-if-statement> evaluates its <boolean-expression> and if the expression's data value is the data value **True**, it executes the <list-or-label> element that follows the keyword **Then**. If the expression's data value is the data value **False**, it executes the <list-or-label> following the keyword **Else**.

- A <list-or-label> is executed by executing each of its constituent <same-line-statement> elements in sequential order until either the last contained <statement> has executed or an executed statement explicitly transfers execution outside of the <list-or-label>.

5.4.2.10 Select Case Statement

A <select-case-statement> determines which <statement-block> to execute out of a candidate set.

```
select-case-statement = "Select" "Case" WS select-expression EOS
*[case-clause]
[case-else-clause]
"End" "Select"
case-clause = "Case" range-clause ["," range-clause] EOS statement-block

case-else-clause = "Case" "Else" EOS statement-block
range-clause = expression
range-clause =/  start-value "To" end-value
range-clause =/    ["Is"] comparison-operator expression
start-value = expression
end-value = expression
select-expression = expression

comparison-operator = "=" / ("<" ">" ) / (">" "<") / "<" / ">" / (">" "=") /  ("=" ">") /
("<" "=") / ("=" "<")
```

Runtime Semantics.

- In a <select-case-statement> the <select-expression> is immediately evaluated and then used in the evaluation of each subsequent <case-clause> and <case-else-clause>

- For each <case-clause>, each contained <range-clause> is evaluated in the order defined. If a <range-clause> *matches* a <select-expression>, then the <statement-block> in the <case-clause> is executed. Upon execution of the <statement-block>, execution of the <select-case-statement> immediately completes (and each subsequent <case-clause> is not evaluated).

    - If the <range-clause> is an <expression>, then <expression> is evaluated and its result is compared with the value of <select-expression>. If they are equal, the <range-clause> is considered a *match* for <select-expression>. Any subsequent <range-clause> in the <case-clause> is not evaluated.

    - If the <range-clause> starts with the keyword **Is** or a <comparison-operator>, then the expression "<select-expression> <comparison-operator> <expression>" is evaluated. If the evaluation of this expression returns the data value True, the <range-clause> is considered a *match* for <select-expression>. Any subsequent <range-clause> in the <case-clause> is not evaluated.

    - If the <range-clause> has a <start-value> and an <end-value>, then the expression "((<select-expression>) >= (<start-value>)) And ((<select-expression>) <= (<end-value>))" is evaluated. If the evaluation of this expression returns the data value **True**, the

<range-clause> is considered a *match* for <select-expression>. Any subsequent <range-clause> in the <case-clause> is not evaluated.

- If evaluation of each <range-clause> in each <case-clause> results in no *match*, the <statement-block> within <case-else-clause> is executed. If <select-expression> is the data value **Null**, only the <statement-block> within <case-else-clause> is executed.
- If a <goto-statement> defined outside the <select-case-statement> causes a <statement> within a <statement-block> to be executed, none of <select-expression>, <case-clause>, or <range-clause are evaluated. A <goto-statement> can also cause execution to leave the <statement-block>. If a later <goto-statement> causes execution to re-enter the <statement-block>, the behavior is as specified by the rules defined for the execution of a <statement-block> within a <select-case-statement>.

5.4.2.11 Stop Statement

```
stop-statement = "Stop"
```

Runtime Semantics.

- A <stop-statement> suspends execution of the VBA program in an implementation-defined manner. Whether or not execution can be resumed is implementation-dependent.
- Subject to possible implementation-defined external interventions, all variables maintain their state if execution resumes.

5.4.2.12 GoTo Statement

```
goto-statement = (("Go" "To") / "GoTo") statement-label
```

Static Semantics.

- A procedure containing a <goto-statement> MUST contain exactly one <statement-label-definition> with the same <statement-label> as the <statement-label> defined in the <goto-statement>.

Runtime Semantics.

- A <goto-statement> causes execution to branch to the <statement> immediately following the <statement-label-definition> for <statement-label>.
- If the <statement-label> is the same as the <end-label> of lexically enclosing procedure declaration execution of the current <procedure-body> immediately completes as if statement execution had reached the end of the <procedure-body> element's contained <statement-block>.

5.4.2.13 On...GoTo Statement

```
on-goto-statement = "On" expression "GoTo" statement-label-list
```

Static Semantics.

- A procedure MUST contain exactly one <statement-label-definition> for each <statement-label> in a <statement-label-list>.

Runtime Semantics.

- Let *n* be the value of the evaluation of <expression> after having been Let-coerced to declared type **Integer**.

  If *n* is zero, or greater than the number of <statement-label> defined in <statement-label-list>, then execution of the <on-goto-statement> immediately completes.

- If *n* is negative or greater than 255, an error occurs (number 5, "Invalid procedure call or argument").

- Execution branches to the <statement-label-definition> for the *n*'th <statement-label> defined in <statement-label-list>.

- If the *n*'th <statement-label> defined in <statement-label-list> is the same as the <end-label> of the lexically enclosing procedure declaration, execution of the current <procedure-body> immediately completes as if statement execution had reached the end of the <procedure-body> element's contained <statement-block>.

5.4.2.14 GoSub Statement

```
gosub-statement = (("Go" "Sub") / "GoSub") statement-label
```

Static Semantics.

- A procedure containing a <gosub-statement> MUST contain exactly one <statement-label-definition> with the same <statement-label> as the <statement-label> defined in the <gosub-statement>.

Runtime Semantics.

- A <gosub-statement> causes execution to branch to the <statement> immediately following the <statement-label-definition> for <statement-label>. Execution continues until the procedure exits or a <return-statement> is encountered.

- If the <statement-label> is the same as the <end-label> of lexically enclosing procedure declaration execution of the current <procedure-body> immediately completes as if statement execution had reached the end of the <procedure-body> element's contained <statement-block>.

- Each invocation of a procedure creates its own *GoSub Resumption List* that tracks execution of each <gosub-statement> and each <return-statement> within that procedure in a last-in-first-out (LIFO) manner. Execution of a GoSub statement adds an entry for the current <gosub-statement> to the current procedure's GoSub Resumption List.

5.4.2.15 Return Statement

```
return-statement = "Return"
```

Runtime Semantics.

- A <return-statement> causes execution to branch to the <statement> immediately following the current procedure's *GoSub Resumption List*'s most-recently-inserted <gosub-statement>.

    If the current procedure's GoSub Resumption List is empty, an error occurs (number 3, "Return without GoSub").

5.4.2.16 On...GoSub Statement

```
on-gosub-statement = "On" expression "GoSub" statement-label-list
```

Static Semantics.

- A procedure MUST contain exactly one <statement-label-definition> for each <statement-label> in a <statement-label-list>.

Runtime Semantics.

- Let *n* be the value of the evaluation of <expression> after having been Let-coerced to the declared type **Integer**.
- If *n* is zero, or greater than the number of <statement-label> defined in <statement-label-list>, then execution of the <on-gosub-statement> immediately completes.
- If *n* is negative or greater than 255, an error occurs (number 5, "Invalid procedure call or argument").
- Execution branches to the <statement-label-definition> for the *n*'th <statement-label> defined in <statement-label-list>.
- If the *n*'th <statement-label> defined in <statement-label-list> is the same as the <end-label> of lexically enclosing procedure declaration execution of the current <procedure-body> immediately completes as if statement execution had reached the end of the <procedure-body> element's contained <statement-block>.

5.4.2.17 Exit Sub Statement

```
exit-sub-statement = "Exit" "Sub"
```

Static Semantics.

- An <exit-sub-statement> MUST be lexically contained inside the <procedure-body> of a <subroutine-declaration>.

Runtime Semantics.

- If the <statement-block> causes execution of an <exit-sub-statement>, execution of the procedure and of this statement immediately completes. No other statements following the <exit-sub-statement> in the procedure are executed.

5.4.2.18 Exit Function Statement

exit-function-statement = "Exit" "Function"

Static Semantics.

An <exit-function-statement> MUST be lexically contained inside the <procedure-body> of a <function-declaration>.

Runtime Semantics.

- If the <statement-block> causes execution of an <exit-function-statement>, execution of the procedure and of this statement immediately completes. No other statements following the <exit-function-statement> in the procedure are executed.

5.4.2.19 Exit Property Statement

exit-property-statement = "Exit" "Property"

Static Semantics.

- An <exit-property-statement> MUST be lexically contained inside the <procedure-body> of a property declaration.

Runtime Semantics.

- If the <statement-block> causes execution of an <exit-function-statement>, execution of the procedure and of this statement immediately completes. No other statements following the <exit-property-statement> in the procedure are executed.

5.4.2.20 RaiseEvent Statement

A <raiseevent-statement> invokes a set of procedures that have been declared as *handlers* for a given event.

```
raiseevent-statement = "RaiseEvent" IDENTIFIER ["(" event-argument-list ")"]
event-argument-list = [event-argument *("," event-argument)]
event-argument = expression
```

Static Semantics.

- A <raiseevent-statement> MUST be defined inside a procedure which is contained in a class module.
- <IDENTIFIER> MUST be the name of an event defined in the enclosing class module.
- The referenced event's parameter list MUST be compatible with the specified argument list according to the rules of procedure invocation. For this purpose, all parameters and arguments are treated as positional.

Runtime Semantics.

- The procedures which have been declared as event handlers for the event are invoked in the order in which their *WithEvents variables* were initialized, passing each <event-argument> as a positional argument in the order they appeared from left to right. Assigning to a *WithEvents variable* disconnects all event handlers that it previously pointed to, and causes the variable to move to the end of the list. When an event is raised, the most-recently assigned *WithEvents variable's* event-handling procedures will be the last to be executed.

- If an <positional-param> for the event is declared as *ByRef*, then after each invocation of the procedure, the next invocation's corresponding <event-argument> is initialized to the value that the parameter last contained inside its most recent procedure invocation.
- Any runtime errors which occur in these procedures are handled by that procedure's error-handling policy. If the invoked procedure's error-handling policy is to use the error-handling policy of the procedure that invoked it, the effect is as if the invoked procedure were using the default error-handling policy. This effectively means that errors raised in the invoked procedure can only be handled in the procedure itself.
- If an unhandled error occurs in an invoked procedure, no further event handlers are invoked.

5.4.2.21 With Statement

A <with-statement> assigns a given expression as the active *With block variable* within a statement block.

```
with-statement = "With" expression EOS statement-block "End" "With"
```

Static semantics.

- A <with-statement> is invalid if the declared type of <expression> is not a UDT, a named class, **Object** or **Variant**.
- The **With** block variable is classified as a variable and has the same declared type as <expression>.
- If <expression> is classified as a variable, that variable is the *With block variable* of the <statement-block>.

Runtime semantics.

- If <expression> is classified as a value, property, function, or unbound member:
  - <expression> is evaluated as a value expression.
  - If the value type of the evaluated expression is a class, it is **Set**-assigned to an anonymous *With block variable*. Then, <statement-block> is executed. After <statement-block> executes, **Nothing** is assigned to the anonymous *With block variable*.
  - If the value type of evaluated expression is a UDT, it is **Let**-assigned to an anonymous temporary *With block variable*. Then, <statement-block> is executed.
  - An anonymous *with block variable* has procedure extent.

5.4.3 Data Manipulation Statements

Data manipulation statements declare and modify the contents of variables.

```
Data-manipulation-statement = local-variable-declaration / static-variable-declaration /
local-const-declaration / redim-statement / mid-statement /rset-statement / lset-statement /
let-statement / set-statement
```

5.4.3.1 Local Variable Declarations

```
local-variable-declaration = ("Dim" ["Shared"] variable-declaration-list)
static-variable-declaration = "Static" variable-declaration-list
```

The optional **Shared** keyword provides syntactic compatibility with other dialects of the Basic language and/or historic versions of VBA.

Static Semantics.

- The occurrence of the keyword **Shared** has no meaning.
- Each variable defined within a <local-variable-declaration> or <static-variable-declaration> MUST have a variable name that is different from any other variable name, constant name, or parameter name defined in the containing procedure.
- A variable defined within a <local-variable-declaration> or <static-variable-declaration> contained in a <function-declaration> or a <property-get-declaration> MUST NOT have the same name as the containing procedure name.
- A variable defined within a <local-variable-declaration> or <static-variable-declaration> MUST NOT have the same name as an *implicitly declared (*Simple Name Expressions*)* variable within the containing procedure

Runtime Semantics.

- All variables defined by a <static-variable-declaration> have *module extent*.
- All variables defined by a <local-variable-declaration> have *procedure extent*, unless the <local-variable-declaration> is contained within a *static procedure (section 5.3.1.2)*, in which case all the variables have *module extent*.

5.4.3.2 Local Constant Declarations

```
local-const-declaration = const-declaration
```

Static Semantics.

- Each constant defined within a <local-const-declaration> MUST have a *constant name* that is different from any other constant name, variable name, or parameter name defined in the containing procedure.
- A constant defined within a <local-const-declaration> in a <function-declaration> or a

  <property-get-declaration> MUST NOT have the same name as the containing procedure name.
- A constant defined within a <local-const-declaration> MUST NOT have the same name as an implicitly declared variable within the containing procedure.
- All other static semantic rules defined for <const-declaration> apply to <local-const-declaration>.

5.4.3.3 ReDim Statement

```
redim-statement = "Redim" ["Preserve"] redim-declaration-list

redim-declaration-list = redim-variable-dcl *("," redim-variable-dcl)
redim-variable-dcl = redim-typed-variable-dcl / redim-untyped-dcl
redim-typed-variable-dcl = TYPED-NAME dynamic-array-dim
redim-untyped-dcl = untyped-name dynamic-array-clause

dynamic-array-dim = "(" dynamic-bounds-list ")"
dynamic-bounds-list = dynamic-dim-spec *[ "," dynamic-dim-spec ]
dynamic-dim-spec = [dynamic-lower-bound] dynamic-upper-bound
dynamic-lower-bound = integer-expression  "to"
dynamic-upper-bound = integer-expression

dynamic-array-clause = dynamic-array-dim [as-clause]
```

Static Semantics.

- Each <TYPED-NAME> or <untyped-name> is first matched as a *simple name expression* in this context.
- If the name has no matches, then the <redim-statement> is instead interpreted as a <local-variable-declaration> with a <variable-declaration-list> declaring a *resizable array* with the specified name and the following rules do not apply.
- Otherwise, if the name has a match, this match is the *redimensioned variable*.
- A <redim-typed-variable-dcl> has the same static semantics as if the text of its elements were parsed as a <typed-variable-dcl>.
- A <redim-untyped-dcl> has the same static semantics as if the text of its elements were parsed as an <untyped-variable-dcl>.
- The declared type of the *redimensioned variable* MUST be **Variant** or a resizable array.
- Any <as-clause> contained within a <redim-declaration-list> MUST NOT be an <as-auto-object>; it MUST be an <as-type>.
- The *redimensioned variable* might not be a param array.
- A *redimensioned variable* might not be a *with block variable (section 5.4.2.21)*.

Runtime Semantics.

- Runtime Error 13 is raised if the declared type of a *redimensioned variable* is **Variant** and its value type is not an array.
- Each array in a <redim-statement> is resized according to the dimensions specified in its <bounds-list>. Each element in the array is reset to the default value for its data type, unless the word "preserve" is specified.
- If the **Preserve** keyword is present, a <redim-statement> can only change the *upper bound* of the last *dimension* of an *array* and the number of *dimensions* might not be changed. Attempting to change the *lower bound* of any *dimension*, the *upper bound* of any *dimension* other than the last *dimension* or the number of *dimensions* will result in Error 9 (“Subscript out of range”).
- If a <redim-statement> containing the keyword **Preserve** results in more elements in a dimension, each of the extra elements is set to its *default data value*.

- If a <redim-statement> containing the keyword **Preserve** results in fewer elements in a dimension, the data value of the elements at the indices which are now outside the array's bounds are discarded. Each of these discarded elements is set to its default data value before resizing the array.
- If the *redimensioned variable* was originally declared as an *automatic instantiation variable (section 2.5.1)*, each dependent variable of the *redimensioned variable* remains an *automatic instantiation variable* after execution of the <redim-statement>.
- If the *redimensioned variable* is currently locked by a ByRef formal parameter runtime Error 10 is raised.

5.4.3.4 Erase Statement

An *erase-statement* reinitializes the elements of a *fixed-size array* to their *default values*, and removes the *dimensions* and data of a *resizable array* (setting it back to its initial state).

```
erase-statement = "Erase" erase-list
erase-list = erase-element *[ "," erase-element]
erase-element = l-expression
```

Static Semantics.

- An <l-expression> that is an <erase-element> MUST be classified as a variable, property, function or unbound member.
- If the <l-expression> is classified as a variable it might not be a *With block variable (section 5.4.2.21)* or *param array*.
- The declared type of each <l-expression> MUST be either an array or **Variant**.

Runtime Semantics.

- Runtime error 13 (Type mismatch) is raised if the declared type of an <erase-element> is **Variant** and its value type is not an array.
- For each <erase-element> whose <l-expression> is classified as a variable:
  - If the declared type of an <erase-element> is resizable array or the declared type is **Variant** and the data value of the associated variable is an array, this data value is set to be an empty array with the same element type.
  - If the declared type of an <erase-element> is fixed size array every dependent variable of the associated array value variable is reset to standard initial value of the declared array element type.

5.4.3.5 Mid/MidB/Mid$/MidB$ Statement

```
mid-statement = mode-specifier "(" string-argument "," start ["," length] ")" "=" expression

mode-specifier = ("Mid" / "MidB" / "Mid$" / "MidB$")
string-argument = bound-variable-expression
start = integer-expression
length = integer-expression
```

Static Semantics.

- The declared type of <string-argument> MUST be **String** or **Variant**.

Runtime Semantics.

- If the value of <start> is less than or equal to 0 or greater than the length of <string-argument>, or if <length> is less than 0, runtime error 5 (Invalid procedure call or argument) is raised.
- The data value of <string-argument> MUST be Let-coercible to **String**.
- Let *v* be the data value that results from Let-coercing the data value of the evaluation of <expression> to the declared type **String**.
- The new data value of the variable is identical to *v* except that a span of characters is replaced as follows:
  - If <mode-specifier> is "Mid" or "Mid$":
    - The first character to replace is the character at the 1-based position *n* within <string-argument>, where *n* = <start>. Starting at the first character to replace, the next *x* characters within <string-argument> are replaced by the first x characters of *v*, where *x* = the least of the following: <length>, the number of characters in <string-argument> after and including the first character to replace, or the number of characters in *v*.
  - If <mode-specifier> is "MidB" or "MidB$":
    - The first character to replace is the character at the 1-based position *n* within <string-argument>, where *n* = <start>. Starting at the first byte to replace, the next *x* bytes within <string-argument> are replaced by the first x bytes of *v*, where *x* = the least of the following: <length>, the number of bytes in <string-argument> after and including the first byte to replace, or the number of bytes in *v*.

5.4.3.6 LSet Statement

lset-statement = "LSet" bound-variable-expression "=" expression

Static Semantics.

- The declared type of <bound-variable-expression> MUST be **String**, **Variant**, or a UDT.

Runtime Semantics.

- The value type of <bound-variable-expression> MUST be **String** or a UDT.
- If the value type of <bound-variable-expression> is **String**:
  - Let *qLength* be the number of characters in the data value of <bound-variable-expression>.
  - Let *e* be the data value of <expression> Let-coerced to declared type **String.** o Let *eLength* be the number of characters in *e*.
  - If *eLength* is less than *qLength*:
    - The **String** data value that is the concatenation of *e* followed by (*qLength – eLength)* space characters (U+0020) is Let-assigned into <bound-variable-expression>.

- Otherwise:
  - The **String** data value this is the initial *qLength* characters of *e* are Let-assigned into <bound-variable-expression>.

- If the value type of <bound-variable-expression> is a UDT:
  - The data in <expression> (as stored in memory in an implementation-defined manner) is copied into <bound-variable-expression> variable in an implementation-defined manner.

5.4.3.7 RSet Statement

```
rset-statement = "RSet" bound-variable-expression "=" expression
```

Static Semantics.

- The declared type of <bound-variable-expression> MUST be **String** or **Variant**.

Runtime Semantics.

- The value type of <bound-variable-expression> MUST be String.
- Let *qLength* be the number of characters in the data value of <bound-variable-expression>.
- Let *eLength* be the number of characters in the data value of <expression>
- If the number of characters in <expression> is less than the number of characters in the data value of <bound-variable-expression>:
  - The data value of (*qLength – eLength)* spaces followed by the data value of <expression> is Let-coerced into <bound-variable-expression>.
- Otherwise:
  - The data value of the first *qLength* characters in <expression> are Let-coerced into <bound-variable-expression>.

5.4.3.8 Let Statement

A *let statement* performs ***Let****-assignment* of a non-object value. The **Let** keyword itself is optional and can be omitted.

```
let-statement = ["Let"] l-expression "=" expression
```

Static Semantics.

This statement is invalid if any of the following is true:

- <expression> cannot be *evaluated to a simple data value (section 5.6.2.2)*.
- <l-expression> is classified as something other than a value, variable, property, function or unbound member.
- <l-expression> is classified as a value and the declared type of <l-expression> is any type except a class or **Object**.

- <l-expression> is classified as a variable, the declared type of <l-expression> is any type except a class or **Object**, and a **Let** coercion from the declared type of <expression> to the declared type of <l-expression> is invalid.
- <l-expression> is classified as a property, does not refer to the enclosing procedure, and any of the following is true:
  - <l-expression> has no accessible **Property Let** or **Property Get**.
  - <l-expression> has an inaccessible **Property Let**.
  - <l-expression> has an accessible **Property Let** and a **Let** coercion from the declared type of <expression> to the declared type of <l-expression> is invalid.
  - <l-expression> has no **Property Let** at all and does have an accessible **Property Get** and the declared type of <l-expression> is any type except a class or **Object** or **Variant**.
- <l-expression> is classified as a function, does not refer to the enclosing procedure, and the declared type of <l-expression> is any type except a class or **Object** or **Variant**.
- <l-expression> is classified as a property or function, refers to the enclosing procedure, and any of the following is true:
  - The declared type of <l-expression> is any type except a class or **Object**.
  - A **Let**-coercion from the declared type of <expression> to the declared type of <l-expression> is invalid.

Runtime Semantics.

The runtime semantics of **Let**-assignment are as follows:

- If <l-expression> is classified as an unbound member, resolve it first as a variable, property, function or subroutine.
- If the declared type of <l-expression> is any type except a class or **Object**:
- Evaluate <expression> as a simple data value to get an expression value.
  - **Let**-coerce the expression value from its value type to the declared type of <l-expression>. o If <l-expression> is classified as a variable, assign the coerced expression value to <l-expression>.
  - If <l-expression> is classified as a property, and does not refer to an enclosing **Property Get**:
    - If <l-expression> has an accessible **Property Let**, invoke the **Property Let**, passing it any specified argument list, along with the coerced expression value as an extra final parameter.
    - If <l-expression> does not have a **Property Let** and does have an accessible **Property Get**, runtime error 451 (Property let procedure not defined and property get procedure did not return an object) is raised.
    - If <l-expression> does not have an accessible **Property Let** or accessible **Property Get**, runtime error 450 (Wrong number of arguments or invalid property assignment) is raised.
  - If <l-expression> is classified as a property or function and refers to an enclosing **Property Get** or function, assign the coerced expression value to the enclosing procedure's return value.

- If <l-expression> is not classified as a variable or property, runtime error 450 (Wrong number of arguments or invalid property assignment) is raised.

- Otherwise, if the declared type of <l-expression> is a class or **Object**:
  - Evaluate <expression> to get an expression value.
  - If <l-expression> is classified as a value or a variable:
    - If the declared type of <l-expression> is a class with a default property, a **Let**-assignment is performed with <l-expression> being a property access to the object's default property and <expression> being the coerced expression value.
    - Otherwise, runtime error 438 (Object doesn't support this property or method) is raised.
  - If <l-expression> is classified as a property:
    - If <l-expression> has an accessible **Property Let**:
      - **Let**-coerce the expression value from its value type to the declared type of the property.
      - Invoke the **Property Let**, passing it any specified argument list, along with the coerced expression value as the final value parameter.
    - If <l-expression> does not have a **Property Let** and does have an accessible **Property Get**:
      - Invoke the **Property Get**, passing it any specified argument list, getting back an LHS value with the same declared type as the property.
      - Perform a **Let**-assignment with <l-expression> being the LHS value and <expression> being the coerced expression value.
    - Otherwise, if <l-expression> does not have an accessible **Property Let** or accessible **Property Get**, runtime error 438 (Object doesn't support this property or method) is raised.
  - If <l-expression> is classified as a function:
    - Invoke the function, passing it any specified argument list, getting back an LHS value with the same declared type as the property.
    - Perform a **Let**-assignment with <l-expression> being the LHS value and <expression> being the coerced expression value.
  - Otherwise, if <l-expression> is not a variable, property or function, runtime error 450 (Wrong number of arguments or invalid property assignment) is raised.

5.4.3.9 Set Statement

A **Set** statement performs **Set**-assignment of an object reference. The **Set** keyword is not optional and MUST always be specified to avoid ambiguity with **Let** statements.

```
set-statement = "Set" l-expression "=" expression
```

Static Semantics.

This statement is invalid if any of the following is true:

- <expression> cannot be *evaluated to a data value (section 5.6.2.1)*.
- <l-expression> is classified as something other than a variable, property or unbound member.
- **Set**-coercion from the declared type of <expression> to the declared type of <l-expression> is invalid.
- <l-expression> is classified as a property, does not refer to the enclosing procedure, and <l-expression> has no accessible **Property Set**.

Runtime Semantics. The runtime semantics of **Set**-assignment are as follows:

- Evaluate <expression> as a data value to get a value.
- **Set**-coerce this value from its value type to an object reference with the declared type of <l-expression>.
- If <l-expression> is classified as an unbound member, resolve it first as a variable, property, function or subroutine.
- If <l-expression> is classified as a variable:
  - If the variable is declared with the **WithEvents** modifier and currently holds an object reference other than **Nothing**, the variable's *event handlers* are detached from the current object reference and no longer handle this object's events.
  - Assign the coerced object reference to the variable.
  - If the variable is declared with the **WithEvents** modifier and the coerced object reference is not **Nothing**, the variable's event handling procedures are attached to the coerced object reference and now handle this object's events.
- If <l-expression> is classified as a property with an accessible **Property Let**, and does not refer to an enclosing **Property Get**, invoke the **Property Let**, passing it the coerced object reference as the value parameter.
- If <l-expression> is classified as a property or function and refers to an enclosing **Property Get** or function, assign the coerced expression value to the enclosing procedure's return value.
- If <l-expression> is not classified as a variable or property, runtime error 450 (Wrong number of arguments or invalid property assignment) is raised.

5.4.4 Error Handling Statements

Error handling statements control the flow of execution when exception conditions occur.

```
error-handling-statement = on-error-statement / resume-statement / error-statement
```

Runtime Semantics.

- Each invocation of a VBA procedure has an *error-handling policy* which specifies how runtime errors SHOULD be handled.
- When a procedure invocation is created, its *error-handling policy* is initially set to the *Default* policy, unless the procedure was directly invoked from the *host application,* in which case its *error-handling policy* is initially set to *Terminate*.

- The possible values of a procedure's error handling policy and the semantics of each policy are defined by the following table:

Policy Name	Runtime Semantics
Default	Discard the current procedure activation returning the error object and control to the procedure activation that called the current procedure activation. Apply the calling procedures activations e*rror handling policy*.
Resume Next	Continue execution within the same procedure activation with the <statement> that in normal execution order would be executed immediately after the <statement> whose execution caused the error to be raised.
Goto	Set the current procedure activation's *error handling policy* to *Default*. Record as part of the procedure activation the identity of the <statement> whose execution caused the error to be raised. This is called the *fault statement*, and the error which caused the fault is called the *active error*. The execution continues in the current procedure starting at the current procedure activation's *handler label*.
Retry	Continue execution within the same procedure activation starting with the <statement> whose execution caused the error to be raised and clear the *active error*.
Ignore	Use the Error data value of the current error object as the value of the expression in the current procedure activation whose execution caused the error to be raised. Continue execution as if no error had been raised and clear the *active error*.
Terminate	Perform implementation defined error reporting actions terminate execution of the VBA statements. Whether or not and how execution control is returned to the *host application* is implementation specific.

5.4.4.1 On Error Statement

An <on-error-statement> specifies a new error-handling policy for a VBA procedure.

```
on-error-statement = "On" "Error" error-behavior

error-behavior = ("Resume" "Next") / ("Goto" statement-label)
```

Static Semantics

- The containing procedure MUST contain exactly one <statement-label-definition> with the same <statement-label> as the <statement-label> contained in the <error-behavior> element, unless the <statement-label> is a <line-number-label> whose data value is the **Integer** 0.

Runtime Semantics.

- An <on-error-statement> specifies a new error-handling policy for the current activation of the containing procedure.
- The Err object (section 6.1.3.2) is reset.
- If the <error-behavior> is "Resume Next", the error-handling policy is set to "Resume Next".
- If the <error-behavior> has a <statement-label> that is a <line-number-label> whose data value is the **Integer** data value 0 then the error-handling policy disabled. If the <error-behavior> is any other <statement-label>, then the error-handling policy set to goto the <statement-label>.

5.4.4.2 Resume Statement

resume-statement = "Resume" [("Next" / statement-label)]

Static Semantics.

- If a <statement-label> is specified, the containing procedure MUST contain a <statement-label-definition> with the same <statement-label>, unless <statement-label> is a <line-number-label> whose data value is the **Integer** 0.

Runtime Semantics.

- If there is no *active error*, runtime error 20 (Resume without error) is raised.
- The Err object is reset.
- If the <resume-statement> does not contain the keyword **Next** and either no <statement-label> is specified or the <statement-label> is a <line-number-label> whose data value is the **Integer** 0, then execution continues by re-executing the <statement> in the current procedure that caused the error.
- If the <resume-statement> contains the keyword **Next** or a <statement-label> which is a <line-number-label> whose data value is the **Integer** 0, then execution continues at the <statement> in the current procedure immediately following the <statement> which caused the error.
- If the <resume-statement> contains a <statement-label> which is not a <line-number-label> whose data value is the **Integer** 0, then execution continues at the first <statement> after the <statement-label-definition> for <statement-label>.

5.4.4.3 Error Statement

Error-statement = "Error" error-number

```
error-number = integer-expression
```

Runtime Semantics.

- The data value of <error-number> MUST be a valid *error number* between 0 and 65535, inclusive.
- The effect is as if the Err.Raise method (section 6.1.3.2.1.2) were invoked with the data value of <error-number> pass as the argument to its number parameter.

5.4.5 File Statements

VBA file statements support the transfer of data between VBA programs and external data files.

```
file-statement = open-statement / close-statement / seek-statement / lock-statement / unlock-
statement / line-input-statement / width-statement / write-statement / input-statement / put-
statement / get-statement
```

The exact natures of external data files and the manner in which they are identified is host defined. Within a VBA program, external data files are identified using *file numbers*. A *file number* is an integer in the inclusive range of 1 to 511. The association between external data files and VBA file numbers is made using the VBA Open statement.

VBA file statements support external files using various alternative modes of data representations and structures. Data can be represented using either a textual or binary representation. External file data can be structured as fixed length records, variable length text lines, or as unstructured sequences of characters or bytes. The external encoding of character data is host-defined.

VBA defines three modes of interacting with files: *character mode, binary mode* and *random mode*. In *character mode*, external files are treated as sequences of characters, and data values are stored and accessed using textual representations of the values. For example, the integer value 123 would be literally represented in a file as the character 1, followed by the character 2, followed by the character 3.

Character mode files are divided into *lines* each of which is terminated by an implementation dependent *line termination sequence* consisting of one or more characters that marks the end of a line. For output purposes a character mode file can have a *maximum line width* which is the maximum number of characters that can be output to a single *line* of the file. Within a *line*, characters positions are identified as numbered *columns*. The left-most *column* of a line is *column* 1. A *line* is also logically divided into a sequence of fourteen-character wide *print zones*.

In *binary mode*, data values are stored and accessed using an implementation-defined binary encoding. For example, the integer value 123 would be represented using its implementation-defined binary representation. An example of this would be as a four byte binary twos-complement integer in little endian order.

In *random mode*, values are represented in a file the same way as *character mode*, but instead of being accessed as a sequential data stream, files opened in *random mode* are dealt with one *record* at a time. A *record* is a fixed size structure of binary-encoded data values. Files in *random mode* contain a series of *records*, numbered 1 through n.

A *file-pointer-position* is defined as the location of the next *record* or byte to be used in a read or write operation on a *file number*. The *file-pointer-position* of the beginning of a fine is 1. For a *character mode* file, the *current line* is the *line* of the file that contains the current *file-pointer-position*. The *current line position* is 1 plus the current *file-pointer-position* minus the *file-pointer position* of the first character of the *current line*.

5.4.5.1 Open Statement

An <open-statement> associates a file number with an external data file and establishes the processing modes used to access the data file.

```
open-statement = "Open" path-name [mode-clause] [access-clause] [lock] "As" file-number [len-
clause]
```

```
path-name = expression
mode-clause = "For" mode

mode = "Append" / "Binary" / "Input" / "Output" / "Random"
access-clause = "Access" access

access = "Read" / "Write" / ("Read" "Write")
lock = "Shared" / ("Lock"  "Read") / ("Lock" "Write") / ("Lock" "Read" "Write")


len-clause = "Len" "=" rec-length
rec-length = expression
```

Static Semantics.

- If there is no <mode-clause> the effect is as if there were a <mode-clause> where <mode> is keyword **Random**. If there is no <access-clause> the effect is as if there were an <access-clause> where <access> is determined by the value of <mode>, according to the following table:

Value of <mode>	File Access Type	Implied value of <access>
Append	Character	Read Write
Binary	Binary	Read Write
Input	Character	Read
Output	Character	Write
Random	Random	Read Write

- If <mode> is the keyword **Output** then <access> MUST consist of the keyword **Write**. If <mode> is the keyword **Input** then <access> MUST be the keyword **Read**. If <mode> is the keyword **Append** then <access> MUST be either the keyword sequence **Read Write** or the keyword **Write**.
- If there is no <lock> element, the effect is as if <lock> is the keyword **Shared**.
- If no <len-clause> is present, the effect is as if there were a <len-clause> with <rec-length> equal to the **Integer** data value 0.

Runtime Semantics.

- The <open-statement> creates an association between a *file number (section 5.4.5)* specified via <file-number> and an external data file identified by the <path-name>, such that occurrences of that same *file number* as the <file number> in subsequently executed file statements are interpreted as references to the associated external data file. Such a file number for which an external association has been successfully established by an <open-statement> is said to be *currently open*.
- An <open-statement> cannot remap or change the <mode>, <access>, or <lock> of an already in-use <file-number>; the association between integer file number and an external data file remains in effect until they are explicitly disassociated using a <close-statement>.

- If an <open-statement> fails to access the underlying file for any reason, an error is generated.
- The value of <path-name> MUST have a data value that is Let-coercible to the declared type **String**. The coerced **String** data value MUST conform to the implementation-defined syntax for external file identifiers.
- The Let-coerced **String** data value of <path-name> is combined with the current drive value (see the ChDrive function in section 6.1.2.5.2.2) and current directory value in an implementation defined manner to obtain a *complete path specification.*
- If the external file specified by the complete path specification <path-name> does not exist, an attempt is made to create the external file unless <mode> is the keyword **Input**, in which case an error is generated.
- If the file is already opened by another process or the system cannot provide the locks requested by <lock>, then the operation fails and an error (number 70, "Permission denied") is generated. If the file cannot be created, for any reason, an error (number 75, "Path/File access error" is generated.
- An error (number 55, "File already open") is generated if the <file-number> integer value already has an external file association that was established by a previously executed <open-statement>.
- The expression in a <len-clause> production MUST evaluate to a data value that is Let-coercible to declared type **Integer** in the inclusive range 1 to 32,767. The <len-clause> is ignored if <mode> is **Binary**.
- If <mode> is **Append** or **Output**, the path specification MUST NOT identify an external file that currently has a file number association that was established by a previously executed <open-statement>. If an external file has associations with multiple file number associations then the interaction of file statements using the different file numbers is implementation defined. The value of <mode> controls how data is read from, and written to, the file. When <mode> is **Random**, the file is divided into multiple records of a fixed size, numbered 1 through n.

Value of <mode>	Description
Append	Data can be read from the file, and any data written to the file is added at the end
Binary	Data can be read from the file, and any data written to the file replaces old data
Input	Data can only be sequentially read from the file
Output	Data can only be sequentially written to the file
Random	Data can be read from or written to the file in chunks (records) of a certain size

- The <access> element defines what operations can be performed on an open *file number* by subsequently executed file statements. The list of which operations are valid in each combination of <mode> and <access> is outlined by the following table:

Statement/Mode	Append	Binary	Input	Output	Random
Get #	**-**	**R, RW**	**-**	**-**	**R, RW**
Put #	**-**	**RW, W**	**-**	**-**	**RW, W**
Input #	**-**	**R, RW**	**R**	**-**	**-**
Line Input #	**-**	**R, RW**	**R**	**-**	**-**

Statement/Mode	Append	Binary	Input	Output	Random
Print #	**RW, W**	**-**	**-**	**W**	**-**
Write #	**RW, W**	**-**	**-**	**W**	**-**
Seek	**RW, W**	**R, RW, W**	**R**	**W**	**R, RW, W**
Width #	**RW, W**	**R, RW, W**	**R**	**W**	**R, RW, W**
Lock	**RW, W**	**R, RW, W**	**R**	**W**	**R, RW, W**
Unlock	**RW, W**	**R, RW, W**	**R**	**W**	**R, RW, W**

Key:

R The statement can be used on a <file-number> where <access> is Read

W The statement can be used on a <file-number> where <access> is Write

RW The statement can be used on a <file-number> where <access> is Read/Write **-** The statement can never be used in the current mode

- The <lock> element defines whether or not agents external to this *VBA Environment* can access the external data file identified by the complete path specification while the *file number* association established by this <open-statement> is in effect. The nature of such external agents and mechanisms they might use to access an external data file are implementation defined. The exact interpretation of the <lock> specification is implementation defined but the general intent of the possible lock modes are defined by the following table:

Lock Type	Description
Shared	External agents can access the file for read and write operations
Lock Read	External agents cannot read from the file
Lock Write	External agents cannot write to the file
Lock Read Write	External agents cannot open the file

- The value of <rec-length> is ignored when <mode> is **Binary**. If <mode> is **Random**, the value of <rec-length> specifies the sum of the individual sizes of the data types that will be read from the file (in bytes). If <rec-length> is unspecified when <mode> is **Random**, the effect is as if <rec-length> is 128. For all other values of <mode>, <rec-length> specifies the number of characters to read in each individual read operation.
- If <mode> is **Random**, when a file is opened the file-pointer-position points at the first record. Otherwise, the file-pointer-position points at the first byte in the file.

5.4.5.1.1 File Numbers

```
file-number = marked-file-number / unmarked-file-number

marked-file-number = "#" expression
unmarked-file-number = expression
```

Static Semantics.

- The *declared type (section 2.2)* of the <expression> element of a <marked-file-number> or <unmarked-file-number> MUST be a *scalar declared type (section 2.2)*.

Runtime Semantics.

- The *file number value* is the *file number (section 5.4.5)* that is the result of *Let-coercing* the result of evaluating the <expression> element of a <file-number> to *declared type* **Integer**.
- If the <file-number> <expression> element does not evaluate to a value that is *Let-coercible* to declared type **Integer**, error number 52 ("Bad file name or number") is raised.

  If the *file number value* is not in the inclusive range 1 to 511 error number 52 ("Bad file name or number") is raised.

5.4.5.2 Close and Reset Statements

A <close-statement> concludes input/output to a file on the system, and removes the association between a <file-number> and its external data file.

```
close-statement = "Reset" / ("Close" [file-number-list])
file-number-list = file-number *[ "," file-number]
```

Static Semantics.

- If <file-number-list> is absent the effect is as if there was a <file-number-list> consisting of all the integers in the inclusive range of 1 to 511.

Runtime Semantics.

- If any *file number value (section 5.4.5.1.1)* in the <file-number-list> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)* then no action is taken for that *file number*. For each *file number value* from <file-number-list> that is *currently-open*, any necessary implementation-specific processing that can be required to complete previously executed file statements using that *file number* is performed to completion and all implementation-specific locking mechanisms associated with that file number are released. Finally, the association between the *file number* and the external file number is discarded. The *file number* is no longer *currently-open* and can be reused in a subsequently executed <open-statement>.

5.4.5.3 Seek Statement

A <seek-statement> repositions where the next operation on a <file-number> will occur within that file.

```
seek-statement = "Seek" file-number "," position
position = expression
```

Static Semantics:

- The *declared type (section 2.2)* of <position> MUST be a *scalar declared type (section 2.2)*.

Runtime Semantics:

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.

- The *new file position* is the evaluated value of <position> Let-coerced to declared type **Long**.
- An error is raised if the *new file position* is 0 or negative.
- If the <open-statement> for the *file number value* of <file-number> had <mode> **Random**, then the *file-pointer-position*'s location refers to a *record*; otherwise, it refers to a byte.

  If *new file position* is greater than the current size of the file (measured in bytes or *records* depending the <mode> of the <Open-statement> for the *file number value*), the size of the file is extended such that its size is the value *new file position*. This does not occur for files whose currently-open <access> is **Read**. The extended content of the file is implementation defined any can be undefined.
- The file-pointer-position of the file is set to *new file position*.

5.4.5.4 Lock Statement

A <lock-statement> restricts which parts of a file can be accessed by external agents. When used without a <record-range>, it prevents external agents from accessing any part of the file.

```
lock-statement = "Lock" file-number [ "," record-range]

record-range = start-record-number / ([start-record-number] "To" end-record-number)
start-record-number = expression
end-record-number = expression
```

Static Semantics:

- The *declared type (section 2.2)* of <start-record-number> and of <end-record-number> MUST be a *scalar declared type (section 2.2)*.
- If there is no <start-record-number> the effect is as if <start-record-number> consisted of the integer number token 1.

Runtime Semantics.

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- If no <record-range> is present the entire file is locked.
- If the *file number value* was opened with <mode> **Input**, **Output**, or **Append**, the effect is as if no <record-range> was present and the entire file is locked.
- The *start record* is the evaluated value of <start-record-number> Let-coerced to declared type **Long**.
- The *end record* is the evaluated value of <end-record-number> Let-coerced to declared type **Long**.
- *Start record* MUST be greater than or equal to 1, and less than or equal to *end record*. If not, an error is raised.
- If the *file number value* was opened with <mode> **Random**, *start record* and *end record* define a inclusive span of records within the external data file associated with that *file number value*. In this case, each record in the span is designated as *locked*.

- If the *file number value* was opened with <mode> **Binary**, both <start-record-number> and <end-record-number> define a byte-position within the external data file associated with that *file number*. In this case, all external file bytes in the range *start record* to *end record* (inclusive), are designated as *locked*.

  Locked files or *locked* records or bytes within a file might not be accessed by other external agents. The mechanism for actually implementing such locks and whether or not a lock can be applied to any specific external file is implementation defined.

- Multiple lock ranges established by multiple lock statements can be simultaneously active for an external data file. A lock remains in effect until it is removed by an <unlock-statement> that specifies the same *file number* as the <lock-statement> that established the lock and which either unlocks the entire file or specifies an <record-range> evaluates to the same *start record* and *end record*. A <close-statement> remove all locks currently established for its *file number value*.

5.4.5.5 Unlock Statement

An <unlock-statement> removes a restriction which has been placed on part of a currently-open file number. When used without a <record-range>, it removes all restrictions on any part of the file.

```
unlock-statement = "Unlock" file-number [ "," record-range]
```

Static Semantics.

- The static semantics for <lock-statement> also apply to <unlock-statement> *Runtime Semantics*.
- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- If no <record-range> is present the entire file is no longer *locked (section 5.4.5.4)*.
- If the *file number value* was opened with <mode> **Input**, **Output**, or **Append**, the effect is as if no <record-range> was present and the entire file is no longer *locked*.
- The *start record* is the evaluated value of <start-record-number> of <record-range> Let-coerced to declared type **Long**.
- The *end record* is the evaluated value of <end-record-number> of <record-range> Let-coerced to declared type **Long**.
- *Start record* MUST be greater than or equal to 1, and less than or equal to *end record*. If not, an error is raised.
- If <record-range> is present, its *start record* and *end record* MUST designate a range that is identical to a *start record* to *end record range* of a previously executed <lock-statement> for the same currently-open file number. If is not the case, an error is raised.
- If the *file number value* was opened with <mode> **Random**, *start record* and *end record* define a inclusive span of records within the external data file associated with that *file number value*. In this case, each record in the span is designated as no longer *locked*.
- If the *file number value* was opened with <mode> **Binary**, both <start-record-number> and <end-record-number> define a byte-position within the external data file associated with that *file number*. In this case, all external file bytes in the range *start record* to *end record* (inclusive), are designated as no longer *locked*.
- If a <record-range> is provided for only the <lock-statement> or the <unlock-statement> designating the same *currently open file number* an error is generated.

5.4.5.6 Line Input Statement

A <line-input-statement> reads in one line of data from the file underlying <marked-file-number>.

```
line-input-statement = "Line"  "Input" marked-file-number "," variable-name

variable-name = variable-expression
```

Static Semantics.

- The <variable-expression> of a <variable-name> MUST be classified as a variable.
- The semantics of <marked-file-number> in this context are those of a <file-number> element that consisted of that same <marked-file-number> element.
- The declared type of a <variable-name> MUST be **String** or **Variant**.

Runtime Semantics.

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- The sequence of bytes starting at the current *file-pointer-position* in the file identified by the *file number value* and continuing through the last byte of the current *line (section 5.4.5)* (but not including the *line termination sequence (section 5.4.5)*) is converted in an implementation dependent manner to a **String** data value.
- If the end of file is reach before finding a line *termination sequence*, the data value is the **String** data value converted from the byte sequence up to the end of the file.
- If the file is empty or there are no characters after *file-pointer-position*, then runtime error 62 ("Input past end of file") is raised.
- The new *file-pointer-position* is equal to the position of the first character after the end of the *line termination sequence*. If the end-of-file was reached the *file-pointer-position* is set to the position immediately following the last character in the file.
- The String data value is Let-assigned into <variable-name>.

5.4.5.7 Width Statement

A <width-statement> defines the maximum number of characters that can be written to a single line in an output file.

```
width-statement = "Width"   marked-file-number   ","  line-width
line-width = expression
```

Static Semantics.

- The semantics of <marked-file-number> in this context are those of a <file-number> element that consisted of that same <marked-file-number> element.
- The *declared type (section 2.2)* of <line-width> MUST be a *scalar declared type (section 2.2)*.

Runtime Semantics.

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- The *line width* is the evaluated value of <line-width> Let-coerced to declared type **Integer**.
- If *Line width* is less than 0 or greater than 255 an error (number 5, "Invalid procedure call or argument") is raised.
- If the *file number value* was opened with <mode> **Binary** or **Random** this statement has no effect upon the file. Otherwise:
  - Each *currently open file numbe*r has an associated *maximum line length (section 5.4.5)* that controls how many characters can be output to a line when using that *file number*. This statement sets the maximum line length of *file number value* to *line width*.
  - If *line width* is 0 then *file number value* is set to have no *maximum line length*.

5.4.5.8 Print Statement

A <print-statement> writes data to the file underlying <marked-file-number>.

```
print-statement = "Print" marked-file-number "," [output-list]
```

Static Semantics.

- The semantics of <marked-file-number> in this context are those of a <file-number> element that consisted of that same <marked-file-number> element.

Runtime Semantics.

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- If <output-list> is not present, the *line termination sequence (section 5.4.5)* is written to the file associated with *file number value* starting at its current *file-pointer-position*. The current *file-pointer-position* is set immediately after the line termination sequence.
- Otherwise, for each <output-item> in <output-list> proceeding in left to right order:
- If <output-clause> consists of an <output-expression>
  1. The <output-expression> is evaluated to produce an *output string value* and characters of the string are written to the file associated with *file number value* starting at its current *file-pointer-position*.
  2. The current *file-pointer-position* now points to the location after the output characters of the string.
  3. If while performing any of these steps the number of characters in the *current line (section 5.4.5)* reaches the *maximum line length (section 5.4.5)* the *line termination sequence* is immediately written and output continues on the next line.
- If <output-clause> consists of a <spc-clause>
  1. If *space count (section 5.4.5.8.1)* is less than or equal to *maximum line length* of the *file number value* or if the *file number value* does not have a *maximum line length,* let *s* be the value of *space count*.

2. Otherwise, *space count* is greater than the *maximum line length*. Let *s* be the value (*space count* modulo *maximum line length*).

3. If the is a *maximum line width* and *s* is greater than *maximum line width* minus *current line position* let *s* equal *s* minus (*maximum line width* minus *current line position*). The *line termination sequence* is immediately written and current *file-pointer-position* is set to beginning of the new line.

4. Write *s* space characters to the file associated with *file number value* starting at its current *file-pointer-position* and set the current *file-pointer-position* to the position following that last such space character.

- If <output-clause> consists of a <tab-clause> that includes a <tab-number-clause> then do the following steps:

  1. If *tab number (section 5.4.5.8.1)* is less than or equal to *maximum line length* of the *file number value* or if the *file number value* does not have a *maximum line length,* let *t* be the value of *tab number*.

  2. Otherwise, *tab number* is greater than the *maximum line length*. Let *t* be the value (*tab number* modulo *maximum line length*).

  3. If *t* less than or equal to the current line position, output the *line termination sequence*. Set the current *file-pointer-position* is set to beginning of the new line.

  4. Write *t* minus *current line position* space characters to the file associated with *file number value* starting at its current *file-pointer-position* and set the current *file-pointer-position* to the position following that last such space character.

- If <output-clause> consists of a <tab-clause> that does not includes a <tab-number-clause> then the current *file-pointer-position* is advanced to the next *print zone (section 5.4.5)* by outputting space characters until (*current line position* modulo 14) equals 1. o If the <char-position> of the <output-item> is ",", the current *file-pointer-position* is further advanced to the next *print zone* by outputting space characters until (modulo 14) equals 1. Note that the *print zone* is advance even if the current file-pointer-position is already at the beginning of a *print zone*.

- If the <char-position> of the last <output-item> is neither a "," or an explicitly occurring ";" the implementation-defined line termination sequence is output and the current *file-position-pointer* is set to the beginning of the new line.

- The *output string value* of an <output-expression> is determined as follows:

  - If the evaluated data value of the <output-expression> is the **Boolean** data value **True**, the output string is "True".

  - If the evaluated data value of the <output-expression> is the **Boolean** data value **False**, the output string is "False".

  - If the evaluated data value of the <output-expression> is the data value **Null**, the output string is "Null".

  - If the evaluated data value of the <output-expression> is an **Error** data value the output string is "Error " followed by the error code Let-coerced to **String**.

  - If the evaluated data value of the <output-expression> is any numeric data value other than a **Date** the output string is the evaluated data value of the <output-expression> Let-coerced to **String** with a space character inserted as the first and the last character of the **String** data value.

- If the evaluated data value of the <output-expression> is a **Date** data value the output string is the data value Let-coerced to **String**.
- Otherwise, the output string is the evaluated data value of the <output-expression> **Let**-coerced to **String**.

5.4.5.8.1 Output Lists

```
output-list = *output-item

output-item = [output-clause] [char-position]

output-clause = (spc-clause / tab-clause / output-expression)
char-position = ( ";" / ",")

output-expression = expression

spc-clause = "Spc" "(" spc-number ")"
spc-number = expression
tab-clause = "Tab" [tab-number-clause]
tab-number-clause = "(" tab-number ")"
tab-number = expression
```

Static Semantics.

- If an <output-item> contains no <output-clause>, the effect is as if the <output-item> contains an <output-clause> consisting of the zero-length string "".
- If <char-position> is not present, then the effect is as if <char-position> were ";".
- The *declared type (section 2.2)* of <spc-number> and of <tab-number> MUST be a *scalar declared type (section 2.2)*.

Runtime Semantics.

- The *space count* of a <spc-clause> is the larger of 0 and the evaluated value of its <spc-number> *Let-coerced* to declared type **Integer**.
- The *tab number* of a <tab-clause> that includes a <tab-number-clause> is the larger of 1 and the evaluated value of its <tab-number> *Let-coerced* to declared type **Integer**.

5.4.5.9 Write Statement

A <write-statement> writes data to the file underlying <marked-file-number>.

```
write-statement = "Write" marked-file-number "," [output-list]
```

Static Semantics.

- The semantics of <marked-file-number> in this context are those of a <file-number> element that consisted of that same <marked-file-number> element.
- If a <write-statement> contains no <output-list>, the effect is as if <write-statement> contains an <output-list> with an <output-clause> of "" (a zero-length string), followed by a <char-position> of ",".

Runtime Semantics.

- An error (number 52, "Bad file name or number") is raised if the file number value (section 5.4.5.1.1) of <file-number> is not a currently-open (section 5.4.5.1) file number (section 5.4.5).
- If <output-list> is not present, the implementation-defined line termination sequence is written to the file associated with file number value starting at its current file-pointer-position. The current file-pointer-position is set immediately after the line termination sequence.
- Otherwise, for each <output-item> in <output-list> proceeding in left to right order:
  - If <output-clause> consists of an <output-expression>:
    1. The <output-expression> is evaluated to produce an *output string value* and characters of the string are written to the file associated with *file number value* starting at its current *file-pointer-position*.
    2. Write a comma character to the file unless this is the final <output-clause> and its <char-position> is neither a "," or an explicitly occurring ";".
    3. Advance the current *file-pointer-position to immediately follow the last output character.*
    4. If while performing any of these steps the number of characters in the *current line (section 5.4.5)* reaches the *maximum line length (section 5.4.5)* the *line termination sequence* is immediately written and output continues on the next line.
  - If <output-clause> consists of a <spc-clause>:
    1. If *space count (section 5.4.5.8.1)* is less than or equal to *maximum line length* of the *file number value* or if the *file number value* does not have a *maximum line length,* let *s* be the value of *space count*.
    2. Otherwise, *space count* is greater than the *maximum line length*. Let *s* be the value (*space count* modulo *maximum line length*).
    3. If the is a *maximum line width* and *s* is greater than *maximum line width* minus *current line position* let *s* equal *s* minus (*maximum line width* minus *current line position*). The *line termination sequence* is immediately written and current *file-pointer-position* is set to beginning of the new line.
    4. Write *s* space characters to the file associated with *file number value* starting at its current *file-pointer-position* and set the current *file-pointer-position* to the position following that last such space character.
    5. If the <char-position> element is a "," write a comma character to the file and advance the current file-pointer-position.
    6. If while performing any of these steps the number of characters in the *current line (section 5.4.5)* reaches the *maximum line length (section 5.4.5)* the *line termination sequence* is immediately written and output continues on the next line.
  - If <output-clause> consists of a <tab-clause> that includes a <tab-number-clause> then do the following steps:
    1. If *tab number (section 5.4.5.8.1)* is less than or equal to *maximum line length* of the *file number value* or if the *file number value* does not have a *maximum line length,* let *t* be the value of *tab number*.
    2. Otherwise, *tab number* is greater than the *maximum line length*. Let *t* be the value (*tab number* modulo *maximum line length*).
    3. If *t* less than or equal to the current line position, output the *line termination sequence*. Set the current *file-pointer-position* is set to beginning of the new line.

4. Write *t* minus *current line position* space characters to the file associated with *file number value* starting at its current *file-pointer-position* and set the current *file-pointer-position* to the position following that last such space character.

5. If the <char-position> element is a "," write a comma character to the file and advance the current *file-pointer-position*.

6. If while performing any of these steps the number of characters in the *current line (section 5.4.5)* reaches the *maximum line length (section 5.4.5)* the *line termination sequence* is immediately written and output continues on the next line.

- Otherwise, <output-clause> consists of a <tab-clause> that does not includes a <tab-number-clause> so do the following steps:

  1. Write a comma character and advance the current *file-pointer-position*.

  2. If the <char-position> element is a "," write a comma character to the file and advance the current *file-pointer-position*.

  3. If while performing any of these steps the number of characters in the *current line (section 5.4.5)* reaches the *maximum line length (section 5.4.5)* the *line termination sequence* is immediately written and output continues on the next line.

- If the <char-position> of the last <output-item> is neither a "," nor an explicitly occurring ";" the implementation-defined line termination sequence is output and the current file-position-pointer is set to the beginning of the new line.

- The output string value of an <output-expression> is determined as follows:

  - If the evaluated data value of the <output-expression> is the **Boolean** data value **True**, the output string is ""#TRUE#".

  - If the evaluated data value of the <output-expression> is the **Boolean** data value **False**, the output string is "#FALSE#".

  - If the evaluated data value of the <output-expression> is the data value **Null**, the output string is "#NULL#".

  - If the evaluated data value of the <output-expression> is an **Error** data value the output string is "#ERROR " followed by the error code Let-coerced to **String** followed by the single character "#".

  - If the evaluated data value of the <output-expression> is a **String** data value the output string is the data value of the String data element with surrounding double quote (U+0022) characters.

  - If the evaluated data value of the <output-expression> is any numeric data value other than a **Date** the output string is the evaluated data value of the <output-expression> Let-coerced to **String** ignoring any implementation dependent locale setting and using "." as the decimal separator.

  - If the evaluated data value of the <output-expression> is a **Date** data value the output string is a **String** data value of the form #yyyy-mm-dd hh:mm:ss#. Hours are specified in 24-hour form. If both the date is 1899-12-30 and the time is 00:00:00 only the date portion is output. Otherwise if the date is 1899-12-30 only the time portion is output and if the time is 00:00:00 only the date portion is output.

  - Otherwise, the output string is the evaluated data value of the <output-expression> Let-coerced to **String** with the data value of the string surrounded with double quote (U+0022) characters.

5.4.5.10 Input Statement

An <input-statement> reads data from the file underlying <marked-file-number>.

```
input-statement = "Input" marked-file-number "," input-list

input-list = input-variable *[ ","  input-variable]
input-variable = bound-variable-expression
```

Static Semantics.

- The semantics of <marked-file-number> in this context are those of a <file-number> element that consisted of that same <marked-file-number> element.
- The <bound-variable-expression> of an <input-variable> MUST be classified as a variable.
- The declared type of an <input-variable> MUST NOT be **Object** or a specific name class.

Runtime Semantics.

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- An <input-statement> reads data (starting from the current file-pointer-position) into one or more variables. Characters are read using the *file number value* until a non-whitespace character is encountered. These whitespace characters are discarded, and the file-pointer-position now points at the first non-whitespace character.
- The following process occurs for each <input-variable> in <input-list>:
  - If the declared type of <input-variable> is String then it is assigned a sequence of characters which are read from the file, defined as:
    1. If the first character read is a DQUOTE then the sequence of characters is a concatenation of all characters read from the file until a DQUOTE is encountered; neither DQUOTE is included in the sequence of characters. The file-pointer-position now points at the character after the second DQUOTE. The beginning and ending DQUOTEs are not included in the String assigned to <input-variable>.
    2. If the first character read is *not* a DQUOTE then the sequence of characters is a concatenation of all characters read from the file until a "," is encountered. The "," is not included in the sequence of characters. The file-pointer-position now points at the character after the ",".
  - If the declared type of <input-variable> is Boolean then it is assigned the value false, unless the sequence of characters read are "#TRUE#". If the sequence of characters is numeric an "Overflow" error is generated (error number 6). The file-pointer-position now points at the character after the second "#". o If the declared type of <input-variable> is Date then a sequence of characters is read from the file, according to the following rules:
    1. If the first character at file-pointer-position is "#", then characters are read until a second "#" is encountered. At this point the concatenated String of characters is Let-coerced into <input-variable>.
    2. If the first character at file-pointer-position is not "#", then error 6 ("Overflow") is generated.

- If the sequence of characters are all numbers or characters which are valid in a VBA number (in other words, ".", "e", "E", "+", "-") then the characters are concatenated together into a string and Let-coerced into the declared type of <input-variable>. The file-pointer-position now points at the first non-numeric character it encountered.
- If the sequence of characters is surrounded by DQUOTEs and the declared type of <input-variable> is not String or Variant, then <input-variable> is set to its default value.
- In this case the file-pointer-position now points at the first character after the second DQUOTE. If this character is a "," then the file-pointer-position advances one more position.
- If the sequence of characters read from the file are "#NULL#" then the Null value is **Let**-coerced into <input-variable>. If the sequence of characters read from the file are "#ERROR " followed by a number followed by a "#" then the error number value is **Let**-coerced into <input-variable>.
- If one of the operations described in this section causes more characters to be read from the file but file-pointer-position is already pointing at the last character in the file, then an "Input past end of file" error is raised (error number 62).

- Each <input-variable> defined in <input-list> is processed in the order specified; if the same underlying variable is specified multiple times in <input-list>, its value will be the one assigned to the last <input-variable> in <input-list> that represents the same underlying variable.

5.4.5.11 Put Statement

```
put-statement = "Put" file-number ","[record-number] "," data

record-number = expression
data = expression
```

Static Semantics.

- The declared type of a <data> expression MUST NOT be **Object**, a named class, or a UDT whose definition recursively includes such a type.
- If no <record-number> is specified, the effect is as if <record-number> is the current *file-pointer-position*.

Runtime Semantics.

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- The value of <record-number> is defined to be the value of <record-number> after it has been Let-coerced to a Long.
- If the <mode> for <file-number> is Binary:
  - The file-pointer-position is updated to be exactly <record-number> number of bytes from the start of the file underlying <marked-file-number>.
  - The value of <data> is written to the file at the current file-pointer-position (according to the rules defined in the Variant Data File Type Descriptors and Binary File Data Formats tables).
  - If <data> is a UDT, then the value of each member of the UDT is written to the file at the current file-pointer-position (according to the rules defined in the Variant Data File Type

Descriptors and Binary File Data Formats tables), in the order in which the members are declared in the UDT.

- If the <mode> for <file-number> is Random:
  - The file-pointer-position is updated to be exactly (<record-number> * <rec-length>) number of bytes from the start of the file underlying <marked-file-number>. o The value of <data> is written to the file at the current file-pointer-position (according to the rules defined in the Variant Data File Type Descriptors and Binary File Data Formats tables).
  - If <data> is a UDT, then the value of each member of the UDT is written to the file at the current file-pointer-position (according to the rules defined in the Variant Data File Type Descriptors and Binary File Data Formats tables), in the order in which the members are declared in the UDT.
  - If the number of bytes written is less than the specified <rec-length> (see section 5.4.5.1) then the remaining bytes are written to the file are undefined. If the number of bytes written is more than the specified <rec-length>, an error is generated (#59, "Bad record length").

When outputting a variable whose declared type is Variant, a two byte *type descriptor* is output before the actual value of the variable.

Variant Kind	Type Descriptor Byte 1	Type Descriptor Byte 2
Unknown	ERROR	-
User Defined Type	ERROR	-
Object	ERROR	-
Data value Empty	00	00
Data value Null	01	00
Integer	02	00
Long	03	00
Single	04	00
Double	05	00
Currency	06	00
Date	07	00
String	08	00
Error	10	00
Boolean	11	00
Decimal	14	00
LongLong	20	00

Once the type descriptor has been written to the file (if necessary), the literal value of the variable is output according to the rules described in the following table:

Data Type	Bytes to write to file
Integer	A two byte signed integer output in little-endian form. See _int16 in [MS-DTYP].
Long	A four byte signed integer. See _int32 in [MS-DTYP].
Single	A four byte IEEE floating point value. See float in [MS-DTYP].
Double	An eight byte IEEE double value. See double in [MS-DTYP].
Currency	An eight byte Currency value. See [MS-OAUT] section 2.2.24.
Date	An eight byte Date value. See [MS-OAUT] section 2.2.25.
String	In **random** mode, the first two bytes are the length of the String. If the value is more than 64 kilobytes, then the value of the first two bytes is FF FF. In **binary** mode there is no two-byte prefix, and the String is stored in ANSI form, without NULL termination
Fixed-length String	There is no two-byte prefix, and the String is stored in ANSI form, without NULL termination
Error	The value of the error code. See HRESULT in [MS-DTYP].
Boolean	If the data value of the Boolean is True, then the two bytes are FF FF. Otherwise, the two bytes are 00 00.
Decimal	A 16 bytes Decimal value. See [MS-OAUT] section 2.2.26.

5.4.5.12 Get Statement

```
get-statement = "Get" file-number "," [record-number] "," variable

variable = variable-expression
```

Static Semantics.

- The <variable-expression> of a <variable> MUST be classified as a variable.
- The declared type of a <variable> expression MUST NOT be **Object**, a named class, or a UDT whose definition recursively includes such a type.
- If no <record-number> is specified, the effect is as if <record-number> is the current *file-pointer-position*.

Runtime Semantics:

- An error (number 52, "Bad file name or number") is raised if the *file number value (section 5.4.5.1.1)* of <file-number> is not a *currently-open (section 5.4.5.1) file number (section 5.4.5)*.
- A <get-statement> reads data from an external file and stores it in a variable.
- If the <mode> for <file-number> is Binary:

- The file-pointer-position is updated to be exactly <record-number> number of bytes from the start of the file underlying <marked-file-number>.
- If the declared type of <variable> is **Variant**:
  - Two bytes are read from the file. These two bytes are the *type descriptor* for the data value that follows. The number of bytes to read next are determined based on the type that the *type descriptor* represents , as shown in the Binary File Data Formats table in section 5.4.5.11. If the value type of <variable> is String, then the number of bytes to read is the number of characters in <variable>.
  - Once these bytes have been read from the file, the data value they form is **Let**-coerced into <variable>.
- If the declared type of <variable> is not Variant:
  - Based on the declared type of <variable>, the appropriate number of bytes are read from the file, as shown in the Variant Data File Type Descriptors table in section 5.4.5.11. Once these bytes have been read from the file, the data value they form is Let-coerced into <variable>.

- If the <mode> for <file-number> is Random:
  - The file-pointer-position is updated to be exactly <record-number> * <rec-length> number of bytes from the start of the file underlying <marked-file-number>.
  - If the declared type of <variable> is Variant:
    - Two bytes are read from the file. These two bytes are the *type descriptor* for the data value that follows. The number of bytes to read next are determined based on the type that the *type descriptor* represents, as shown in the Binary File Data Formats table in section 5.4.5.11. Once these bytes have been read from the file, the data value they form is Let-coerced into <variable>.
  - If the declared type of <variable> is String:
    - Two bytes are read from the file. The data value of these two bytes is the number of bytes to read from the file. Once these bytes have been read form the file, the data value they form is Let-coerced into <variable>.
  - If the declared type of <variable> is neither Variant not String:
    - The number of bytes to read from the file is determined by the declared type of <variable>, as shown in the Variant Data File Type Descriptors table in section 5.4.5.11. Once these bytes have been read from the file, the data value they form is Let-coerced into <variable>.

5.5 Implicit coercion

In many cases, values with a given *declared type* can be used in a context expecting a different declared type. The implicit coercion rules defined in this section decide the semantics of such implicit coercions based primarily on the value type of the source value and the declared type of the destination context.

There are two types of *implicit coercion*, ***Let****-coercion (section 5.5.1)* and ***Set****-coercion (section 5.5.2)*, based on the context in which the coercion occurs. Operations that can result in implicit coercion will be defined to use either ***Let****-coercion* or ***Set****-coercion*.

Note that only implicit coercion is covered here. Explicit coercion functions, such as **CInt**, are covered in the *VBA Standard Library* section 6.1.2.3.

The exact semantics of implicit ***Let*** and ***Set*** *coercion* are described in the following sections.

5.5.1 Let-coercion

Let*-coercion* occurs in contexts where non-object values are expected, typically where the declared type of the destination is not a class or **Object**.

Within the following sections, **Decimal** and **Error** are treated as though they are declared types, even though VBA does not define a **Decimal** or **Error** declared type (data values of these value types can be represented only within a declared type of **Variant**). The semantics defined in this section for conversions to **Decimal** and **Error** are used by the definition of **CDec** (section 6.1.2.3.1.6) and **CvErr** (section 6.1.2.3.1.14), respectively.

5.5.1.1 Static semantics

Let-coercion between the following pairs of source declared types or literals and destination declared types is invalid:

Source Declared Type or Literal	Destination Declared Type
Any type	Any fixed-size array
Any numeric type or **Boolean** or **Date**	Resizable **Byte()**
Any type except a non-**Byte** resizable or fixed-size array or **Variant**	Any non-**Byte** resizable array
Any type except a UDT or **Variant**	Any UDT
Any type except **Variant**	Any class or **Object**
Any class which has no accessible default **Property Get** or function, or which has an accessible default **Property Get** or function for which it is statically invalid to **Let**-coerce its declared type to the destination declared type	Any type
Any non-**Byte** resizable or fixed-size array	Resizable array of different element type than source type or any non-array type except **Variant**
Any UDT	Different UDT than source type or any non-UDT type except **Variant**
UDT not imported from external reference or array of UDTs not imported from external reference or array of fixed-length strings	**Variant**
Nothing	Any type except a class or **Object** or **Variant**

It is also invalid to implicitly **Let**-coerce from the **LongLong** declared type to any declared type other than **LongLong** or **Variant**. Such coercions are only valid when done explicitly by use of a ***CType*** explicit coercion function.

5.5.1.2 Runtime semantics

5.5.1.2.1 Let-coercion between numeric types

The most fundamental coercions are conversions from a numeric value type (**Integer**, **Long**, **LongLong**, **Byte**, **Single**, **Double**, **Currency**, Decimal) to a numeric declared type (**Integer**, **Long**, **LongLong**, **Byte, Single**, **Double**, **Currency**).

Numeric value types can be broken down into 3 categories:

- *Integral:* **Integer**, **Long**, **LongLong** and **Byte**
- *Floating-point:* **Single** and **Double**
- *Fixed-point:* **Currency** and **Decimal**

Similarly, *numeric declared types* can be broken down into 3 categories:

- *Integral:* **Integer**, **Long** (including any **Enum**), **LongLong** and **Byte**
- *Floating-point:* **Single** and **Double**
- *Fixed-point:* **Currency** and **Decimal**

The semantics of numeric **Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Any integral type	Any numeric type	If the source value is within the range of the destination type, the result is a copy of the value. Otherwise, runtime error 6 (Overflow) is raised.
Any floating point or fixed point type	Any integral type	If the source value is finite (not positive infinity, negative infinity or NaN) and is within the range of the destination type, the result is the value converted to an integer using *Banker's rounding (section 5.5.1.2.1.1)*. Otherwise, runtime error 6 (Overflow) is raised.
Any integral type	Any numeric type	If the source value is within the range of the destination type, the result is a copy of the value. Otherwise, runtime error 6 (Overflow) is raised.

Source Value Type	Destination Declared Type	Semantics
Any integral type	Any floating point or fixed point type	If the source value is finite (not positive infinity, negative infinity or NaN) and is within the magnitude range of the destination type, the result is the value rounded to the nearest value representable in the destination type using *Banker's rounding*. Otherwise, runtime error 6 (Overflow) is raised. Note that the conversion can result in a loss of precision, and if the value is too small it can become 0.

5.5.1.2.1.1 Banker's rounding

Banker's rounding is a midpoint rounding scheme, also known as round-to-even.

During rounding, ambiguity can arise when the original value is at the midpoint between two potential rounded values. Under *Banker's rounding*, such ambiguity is resolved by rounding to the nearest rounded value such that the least-significant digit is even.

For example, when using *Banker's rounding* to round to the nearest 1, both 73.5 and 74.5 round to 74, while 75.5 and 76.5 round to 76.

5.5.1.2.2 Let-coercion to and from Boolean

When not stored as a **Boolean** value, **False** is represented by 0, and **True** is represented by nonzero values, usually -1.

The semantics of **Boolean Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Boolean	**Boolean**	The result is a copy of the source value.
Boolean	*Any numeric type except* **Byte**	If the source value is **False**, the result is 0. Otherwise, the result is -1.
Boolean	**Byte**	If the source value is **False**, the result is 0. Otherwise, the result is 255.
Any numeric type	**Boolean**	If the source value is 0, the result is **False**. Otherwise, the result is **True**.

5.5.1.2.3 Let-coercion to and from Date

A **Date** value can be converted to or from a *standard **Double** representation* of a date/time, defined as the fractional number of days after 12/30/1899 00:00:00. As **Date** values representing times with

no date are represented as times within the date 12/30/1899, their standard **Double** representation becomes a **Double** value greater than or equal to 0 and less than 1.

The semantics of **Date Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Date	**Date**	The result is a copy of the source date.
Date	Any numeric type or **Boolean**	The result is the standard **Double** representation of the source date **Let**-coerced to the destination type.
Any numeric type or **Boolean**	**Date**	The source value is converted to a **Double** using the **Let**-coercion rules for **Double**. This **Double** representation is then interpreted as a standard **Double** representation of a date/time and converted to a **Date** value. If this date value is within the range of valid **Date** values, the result is the converted date. Otherwise, runtime error 6 (Overflow) is raised.

5.5.1.2.4 Let-coercion to and from String

The formats accepted or produced when coercing number, currency and date values to or from **String** respects host-defined *regional settings*. Excess whitespace is ignored at the beginning or end of the value or when inserted before or after date/time separator characters such as "/" and ":", sign characters such as "+", "-" and the scientific notation character "E".

The semantics of **String Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
String	**String**	The result is a copy of the source string.

String	Any numeric type	The source string is parsed as a numeric-coercion-string using the following case-insensitive, whitespace-sensitive grammar:
		
		`numeric-coercion-string = [WS] [sign [WS]] regionalnumber-string [exponentclause] [WS]`
		
		`exponent-clause = ["e" / "d"] [sign] integer-literal`
		
		`sign = "+" / "-"`
		
		`regional-number-string = <unsigned number or currency value interpreted according to the active host-defined regional settings>`
		
		If the <regional-number-string> can be interpreted as an unsigned number or unsigned currency value according to the active host-defined *regional settings*, an *interpreted value* is determined as follows:

Source Value Type	Destination Declared Type	Semantics
		▪ If the destination type is an integral or fixed-point numeric type, <regional-number-string> is interpreted as an infinite-precision fixed-point numeric value. ▪ Otherwise, if the destination type is a floating-point numeric type, <regional-number-string> is interpreted as an infinite-precision floating-point numeric value. A *scaled value* is then determined as follows: ▪ If <exponent-clause> is not specified, the scaled value is the interpreted value. ▪ Otherwise, if <exponent-clause> is specified, an exponent is determined. The magnitude of the exponent is the value of the <integer-literal> within exponent. If a <sign> is specified, the exponent is given that sign, otherwise the sign of the exponent is positive. The scaled value is the interpreted value multiplied by 10exponent. A *signed value* is then determined as follows: ▪ If a <sign> is specified, the scaled value is given the specified sign. ▪ Otherwise, the sign of the scaled value is positive. The result is then determined from the signed value as follows: ▪ If the destination type is an integral numeric type, and the signed value is within the range of the destination type, the result is the signed value converted to an integer using *Banker's rounding (section 5.5.1.2.1.1)*. ▪ Otherwise, if the destination type is a fixed-point or floating-point numeric type, and the signed value is within the magnitude range of the destination type, the result is the signed value converted to the nearest value that has a representation in the destination type. If the <regional-number-string> could not be interpreted as a number or currency value, runtime error 13 (Type mismatch) is raised. If the value could be interpreted as a number, but was out of the range of the destination type, runtime error 6 (Overflow) is raised. Note that the conversion can result in a loss of precision, and if the value is too small the result can be 0.
String	**Boolean**	If the source string is equal to "True" or "False", case-insensitive, the result is **True** or **False**, respectively. If the source string is equal to "#TRUE#" or "#FALSE#", case-sensitive, the result is **True** or **False**, respectively. The case sensitivity of these string comparisons is not affected by **Option Compare**.

Source Value Type	Destination Declared Type	Semantics
		Otherwise, the result is the source string **Let**-coerced to a **Double** value, which is then **Let**-coerced to a **Boolean** value.
String	**Date**	If the source string can be interpreted as either a date/time, time, or date value (in that precedence order) according to the host-defined *regional settings*, the value is converted to a **Date**. Otherwise, if the source string can be interpreted as a number or currency value according to the host-defined *regional settings*, and the resulting value is within the magnitude range of **Double**, the value is converted to the nearest representable **Double** value, and then this value is **Let**-coerced to **Date**. If this coerced value is within the range of **Date**, the result is the date value. If the source string could not be interpreted as a date/time, time, date, number or currency value, runtime error 13 (Type mismatch) is raised. If the conversion to **Double** resulted in an overflow, runtime error 13 (Type mismatch) is raised instead of the runtime error 6 (Overflow) that would otherwise be raised.

Any numeric type	**String**	The maximum number of integral significant figures that can be output is based on the value type of the source as follows:
		▪ **Single**: 7
		▪ **Double**: 15
		▪ *Any integral or fixed-point type*: Infinite
		
		The number is converted to a string using the following format (note that some host-defined regional number formatting settings, such as custom negative sign symbols and digit grouping, can be ignored):
		
		▪ If the number is 0, the result is the string "0".
		▪ If the number is positive infinity, the result is the string "1.#INF".
		▪ If the number is negative infinity, the result is the string "1.#INF".
		▪ If the number is NaN (not a number), the result is the string "-1.#IND".
		▪ If the number is not 0 and there are less than or equal to the maximum number of integral significant figures in the integer part of the number, **normal notation** is used; for example, -123.45. The resulting string is in the following format:
		  ▪ - if the number is negative
		  ▪ The digits of the integer part of the number with no digit grouping (thousands separators) applied
		  ▪ The host-defined regional decimal symbol (such as **.** or ,) if any fractional digits will be printed next
		  ▪ As many digits as possible of the fractional part of the number such that a maximum of 15 integer and fractional digits are printed total with trailing zeros removed

Source Value Type	Destination Declared Type	Semantics
		▪ If the number is not 0 and there are more than the maximum number of integral significant figures in the integer part of the number, **scientific notation** is used; for example, -1.2345E+2. The number is converted to its equivalent form $s \times 10^e$, where s is the significand (the number scaled such that there is exactly one nonzero digit before the decimal point), and e is the exponent (equal to the number of places the decimal point was moved to form the significand). The resulting string is in the following format: ▪ - if the number is negative ▪ The single digit of the integer part of the significand ▪ The host-defined regional decimal symbol (such as **.** or **,**) if any fractional digits of the **significand** will be printed next ▪ As many digits as possible of the **significand** such that a maximum of 15 integer and **significand** digits are printed total with trailing zeros removed ▪ **E** ▪ **+** or **-** depending on the sign of the exponent ▪ The digits of the exponent Note that the string conversion always interprets the source value as a number, not a currency value, even for fixed-point numeric types such as Currency or Decimal.
Boolean	**String**	If the source value is **False**, the result is "False". Otherwise, the result is "True".
Date	**String**	If the day value of the source date is 12/30/1899, only the date's time is converted to a string according to the host-defined regional **Long Time** format, and the result is this time string. Otherwise, the source date's full date and time value is converted to a string according to the platform's host-defined regional **Short Date** format, and the result is this date/time string. The **Long Time** format represents the platform's standard time format that includes hours, minutes and seconds. The **Short Date** format represents the platform's standard date format where the month, day and year are all expressed in their shortest form (that is, as numbers).

5.5.1.2.5 Let-coercion to String * length (fixed-length strings)

The semantics of **String *** *length* **Let**-coercion depend on the source's value type:

Source Value Type	Destination Declared Type	Semantics
String	**String *** *length*	If the source string has more than *length* characters, the result is a copy of the source string truncated to the first *length* characters. Otherwise, the result is a copy of the source string padded on the right with space characters to reach a total of *length* characters.
Any numeric type, **Boolean** *or* **Date**	**String *** *length*	The result is the source value **Let**-coerced to a **String** value and then **Let**-coerced to a **String *** *length* value.

5.5.1.2.6 Let-coercion to and from resizable Byte()

The semantics of **Byte() Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Byte()	*Resizable* **Byte()**	The result is a copy of the source **Byte** array.
Byte()	**String** *or* **String *** *length*	The binary data within the source **Byte** array is interpreted as if it represents the implementation-defined binary format used to store **String** data. Even if this implementation-defined format includes a prefixed length and/or end marker, these elements are not read from the **Byte** array and MUST instead be inferred from the **String** data. The result is the string produced. This coercion never raises a runtime error. If the byte array is uninitialized, the result is a 0-length string. If binary data in the array cannot be interpreted as a character, or if the character specified is cannot be represented on the current platform, that character is output in the **String** as a **?** character. Any trailing bytes leftover at the end of the byte array that could not be interpreted are discarded.
Byte()	*Any numeric type,* **Boolean** *or* **Date**	The result is undefined.
String	*Resizable* **Byte()**	The result is a copy of the implementation-defined binary data used to store the **String** value, excluding any prefixed length and/or end marker.
Any numeric type, **Boolean** *or* **Date**	*Resizable* **Byte()**	Runtime error 13 (Type mismatch) is raised.

5.5.1.2.7 Let-coercion to and from non-Byte arrays

The semantics of non-**Byte** array **Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
*Any non-***Byte** *array*	*Array with same element type as source type*	The result is a shallow copy of the array. Elements with a value type of a class or **Nothing** are **Set**-assigned to the destination array element and all other elements are **Let**-assigned.
*Any non-***Byte** *array*	*Any other type except* **Variant**	Runtime error 13 (Type mismatch) is raised.
Any numeric type, **Boolean***,* **Date***, or* **String**	*Any fixed-size array or non-***Byte** *resizable array*	Runtime error 13 (Type mismatch) is raised.

5.5.1.2.8 Let-coercion to and from a UDT

The semantics of UDT **Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Any UDT	*Same UDT as source type*	The result is a shallow copy of the UDT. Elements with a value type of a class or **Nothing** are **Set**-assigned to the destination UDT field and all other elements are **Let**-assigned.
Any UDT	*Any other type except* **Variant**	Runtime error 13 (Type mismatch) is raised.
Any numeric type, **Boolean***,* **Date***,* **String** *or array*	*Any UDT*	Runtime error 13 (Type mismatch) is raised.

5.5.1.2.9 Let-coercion to and from Error

The semantics of **Error Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Error	*Any type except a fixed-size array or* **Variant**	Runtime error 13 (Type mismatch) is raised.
Any numeric type, **Boolean***,* **Date***,* **String***, array or UDT*	**Error**	The source value is converted to a **Long** using the **Let-**coercion rules for **Long**. If this **Long** representation is between 0 and 65535, inclusive, the result is an **Error** data value representing the standard error code specified by the **Long** value. Otherwise, runtime error 5 (Invalid procedure call or argument) is raised.

5.5.1.2.10 Let-coercion from Null

The semantics of **Null Let**-coercion depend on the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Null	*Any resizable array or UDT*	Runtime error 13 (Type mismatch) is raised.
Null	*Any other type except a fixed-size array or* **Variant**	Runtime error 94 (Invalid use of Null) is raised.

5.5.1.2.11 Let-coercion from Empty

The semantics of **Empty Let**-coercion depend on the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Empty	*Any numeric type*	The result is 0.
Empty	**Boolean**	The result is **False**.
Empty	**Date**	The result is 12/30/1899 00:00:00.
Empty	**String**	The result is a 0-length string.
Empty	**String *** *length*	The result is a string containing *length* spaces.
Empty	*Any class or*	Runtime error 424 (Object required) is raised.

Source Value Type	Destination Declared Type	Semantics
	Object	
Empty	*Any other type except* **Variant**	Runtime error 13 (Type mismatch) is raised.

5.5.1.2.12 Let-coercion to Variant

The semantics of **Variant Let**-coercion depend on the source's value type:

Source Value Type	Destination Declared Type	Semantics
Any type except a class or **Nothing**	**Variant**	The result is a copy of the source value, **Let**-coerced to the destination declared type.

5.5.1.2.13 Let-coercion to and from a class or Object or Nothing

The semantics of object **Let**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Any class	*Any type*	The result is the simple data value of the object, **Let**-coerced to the destination declared type.
Nothing	*Any type*	Runtime error 91 (Object variable or With block variable not set) is raised.
Any type except a class or **Nothing**	*Any class or* **Object**	Runtime error 424 (Object required) is raised.

5.5.2 Set-coercion

Set-*coercion* occurs in contexts where object values are expected, typically where the declared type of the destination is a class or where the **Set** keyword has been used explicitly.

5.5.2.1 Static semantics

Set-coercion between the following pairs of source declared types and destination declared types is invalid:

Source Declared Type	Destination Declared Type
Any type	*Any type except a class or* **Object** *or* **Variant**

Source Declared Type	Destination Declared Type
Any type except a class or **Object** *or* **Variant**	*Any class or* **Object** *or* **Variant**

5.5.2.2 Runtime semantics

5.5.2.2.1 Set-coercion to and from a class or Object or Nothing

The semantics of object **Set**-coercion depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Any class	*Same class as source type or class implemented by source type or* **Object** *or* **Variant**	The result is a copy of the source object reference. The source and destination now refer to the same object.
Any class	*Different class not implemented by source type*	Runtime error 13 (Type mismatch) is raised.
Nothing	*Any class or* **Object** *or* **Variant**	The result is the **Nothing** reference.

5.5.2.2.2 Set-coercion to and from non-object types

The semantics of non-object **Set**-coercion with the **Set** keyword depend on the source's value type and the destination's declared type:

Source Value Type	Destination Declared Type	Semantics
Any type except a class or **Nothing**	*Any class or* **Object**	Runtime error 424 (Object required) is raised.
Any type except a class or **Nothing**	**Variant**	Runtime error 13 (Type mismatch) is raised.

5.6 Expressions

An *expression* is a hierarchy of values, identifiers and subexpressions that evaluates to a value, or references an entity such as a variable, constant, procedure or type. Besides its tree of subexpressions, an expression also has a *declared type* which can be determined statically, and a *value type* which can vary depending on the runtime value of its values and subexpressions. This section defines the syntax of expressions, their static resolution rules and their runtime evaluation rules.

```
expression = value-expression / l-expression
value-expression = literal-expression / parenthesized-expression / typeof-is-expression /
new-expression / operator-expression
l-expression = simple-name-expression / instance-expression / member-access-expression /
index-expression / dictionary-access-expression / with-expression
```

5.6.1 Expression Classifications

Every expression has one of the following *classifications*:

- A *value expression*. A value expression represents an immutable *data value*, and also has a declared type.
- A *variable expression*. A variable expression references a *variable* declaration, and also has an argument list queue and a declared type.
- A *property expression*. A property expression references a *property*, and also has an argument list queue and a declared type.
- A *function expression*. A function expression references a *function*, and also has an argument list queue and a declared type.
- A *subroutine expression*. A subroutine expression references a *subroutine*, and also has an argument list queue.
- An *unbound member expression*. An unbound member expression references a variable, property, subroutine or function, whose classification or target reference cannot be statically determined, and also has an optional member name and an argument list queue.
- A *project expression*. A project expression references a *project*.
- A *procedural module expression*. A procedural module expression references a *procedural module*.
- A *type expression*. A type expression references a *declared type*.

5.6.2 Expression Evaluation

The *data value* or *simple data value* of an expression can be obtained through the process of *expression evaluation*. Both data values and simple data values represent an immutable value and have a declared type, but simple data values can not represent objects or the value **Nothing**.

5.6.2.1 Evaluation to a data value

Static semantics. The following types of expressions can be evaluated to produce a *data value*:

- An expression classified as a value expression or variable expression can be evaluated as a data value with the same declared type as the expression, based on the following rules:

- If this expression's argument list queue is empty, the declared type of the data value is that of the value.
- Otherwise, if this expression's argument list queue has a first unconsumed argument list (perhaps with 0 arguments):
  - If the declared type of the expression is **Object** or **Variant**, the declared type of the data value is **Variant**.
  - If the declared type of the expression is a specific class:
    - If the declared type of the variable has a public default **Property Get** or function and this default member's parameter list is compatible with this argument list, the declared type of the data value is the declared type of this default member.
    - Otherwise, the evaluation is invalid.
  - If the declared type of the expression is an array type:
    - If the number of arguments specified is equal to the rank of the array, the declared type of the data value is the array's element type.
    - Otherwise, if one or more arguments have been specified and the number of arguments specified is different than the rank of the array, the evaluation is invalid.
  - Otherwise, if the declared type is a type other than **Object**, **Variant**, a specific class or an array type, the evaluation is invalid.
- An expression classified as a property with an accessible **Property Get** or a function can be evaluated as a data value with the same declared type as the property or function.
- An expression classified as an unbound member can be evaluated as a data value with a declared type of **Variant**.

Runtime semantics.

At runtime, the data value's value is determined based on the classification of the expression, as follows:

- If the expression is classified as a value, the data value's value is that of the expression.
- If the expression is classified as an unbound member, the member is resolved as a variable, property, function or subroutine:
  - If the member was resolved as a variable, property or function, evaluation continues as if the expression had statically been resolved as a variable expression, property expression or function expression, respectively.
  - If the member was resolved as a subroutine, the subroutine is invoked with the same target and argument list as the unbound member expression. The data value's value is the value **Empty**.
- If the expression is classified as a variable:
  - If the argument list queue is empty, the data value's value is a copy of the variable's data value.
  - Otherwise, if the argument list queue has a first unconsumed argument list (perhaps empty):
    - If the value type of the expression's target variable is a class:

        - If the declared type of the target is **Variant**, runtime error 9 (Subscript out of range) is raised.
        - If the declared type of the target is not **Variant**, and the target has a public default **Property Get** or function, the data value's value is the result of invoking this default member for that target with this argument list. This consumes the argument list.
        - Otherwise, runtime error 438 (Object doesn't support this property or method) is raised.
    - If the value type of the expression's target is an array type:
        - If the number of arguments specified is equal to the rank of the array, and each argument is within its respective array dimension, the data value's value is a copy of the value stored in the element of the array indexed by the argument list specified. This consumes the argument list.
        - Otherwise, runtime error 9 (Subscript out of range) is raised.
    - Otherwise, if the value type of the expression's target variable is a type other than a class or array type, runtime error 9 (Subscript out of range) is raised.
- If the expression is classified as a property or a function:
    - If the enclosing procedure is either a **Property Get** or a function, and this procedure matches the procedure referenced by the expression, evaluation restarts as if the expression was a variable expression referencing the current procedure's return value.
    - Otherwise, the data value's value is the result of invoking this referenced property's named **Property Get** procedure or function for that target. The argument list for this invocation is determined as follows:
        - If the procedure being invoked has a parameter list that cannot accept any parameters or the argument queue is empty, the procedure is invoked with an empty argument list. In this case, if the argument queue has a first unconsumed argument list and this list is empty, this argument list is consumed.
        - Otherwise, if the procedure being invoked has a parameter list with at least one named or optional parameter, and the argument list queue has a first unconsumed argument list (perhaps empty), the procedure is invoked with this argument list. This consumes the argument list.

5.6.2.2 Evaluation to a simple data value

Static semantics. The following types of expressions can be evaluated to produce a *simple data value*:

- An expression classified as a value expression can be evaluated as a simple data value based on the following rules:
    - If the declared type of the expression is a type other than a specific class, **Variant** or **Object**, the declared type of the simple data value is that of the expression.
    - If the declared type of the expression is **Variant** or **Object**, the declared type of the simple data value is **Variant**.
    - If the declared type of the expression is a specific class:
        - If this class has a public default **Property Get** or function and this default member's parameter list is compatible with an argument list containing 0 parameters, simple data value evaluation restarts as if this default member was the expression.

- An expression classified as an unbound member, variable, property or function can be evaluated as a simple data value if it is both valid to evaluate the expression as a data value, and valid to evaluate an expression with the resulting classification and declared type as a simple data value.

Runtime semantics. At runtime, the simple data value's value and value type are determined based on the classification of the expression, as follows:

- If the expression is a value expression:
  - If the expression's value type is a type other than a specific class or **Nothing**, the simple data value's value is that of the expression.
  - If the expression's value type is a specific class:
    - If the source object has a public default **Property Get** or a public default function, and this default member's parameter list is compatible with an argument list containing 0 parameters, the simple data value's value is the result of evaluating this default member as a simple data value.
    - Otherwise, if the source object does not have a public default **Property Get** or a public default function, runtime error 438 (Object doesn't support this property or method) is raised.
  - If the expression's value type is **Nothing**, runtime error 91 (Object variable or **With** block variable not set) is raised.
- If the expression is classified as an unbound member, variable, property or function, the expression is first evaluated as a data value and then the resulting expression is reevaluated as a simple data value.

5.6.2.3 Default Member Recursion Limits

Evaluation of an object whose default **Property Get** or default function returns another object can lead to a recursive evaluation process if the returned object has a further default member. Recursion through this chain of *default members* can be implicit if evaluating to a simple data value and each default member has an empty parameter list, or explicit if index expressions are specified that specifically parameterize each default member.

An implementation can define limits on when such a recursive default member evaluation is valid. The limits can depend on factors such as the depth of the recursion, implicit vs. explicit specification of empty argument lists, whether members return specific classes vs. returning **Object** or **Variant**, whether the default members are functions vs. **Property Get**s, and whether the expression occurs on the left side of an assignment. The implementation can determine such an evaluation to be invalid statically or can raise error 9 (Subscript out of range) or 13 (Type mismatch) during evaluation at runtime.

5.6.3 Member Resolution

An expression statically classified as a member can be resolved at runtime to produce a variable, property, function or subroutine reference through the process of *member resolution*.

Runtime semantics.

At runtime, an unbound member expression can be resolved as a variable, property, function or subroutine as follows:

- First, the target entity is evaluated to a target data value. Member resolution continues if the value type of the data value is a class or a UDT.
    - If the value type of the target data value is **Nothing**, runtime error 91 (Object variable or With block variable not set) is raised.
    - If the value type of the target data value is a type other than a class, a UDT or **Nothing**, runtime error 424 (Object required) is raised.
- If a member name has been specified and an accessible variable, property, function or subroutine with the given member name exists on the target data value, the member resolves as a variable expression, property expression, function expression or subroutine expression, respectively, referencing the named member with the target data value as the target entity and with the same argument list queue.
- If no member name has been specified, and the target data value has a public default **Property Get** or a public default function, the member resolves as a property expression or function expression respectively, referencing this default member with the target data value as the target entity and with the same argument list queue.
- Otherwise, if no resolution was possible:
    - If the value type of the target entity is a class, runtime error 438 (Object doesn't support this property or method) is raised. o If the value type of the target entity is a UDT, runtime error 461 (Method or data member not found) is raised.

5.6.4 Expression Binding Contexts

An expression can perform name lookup using one of the following *binding contexts*:

- The *default binding context*. This is the binding context used by most expressions.
- The *type binding context*. This is the binding context used by expressions that expect to reference a type or class name.
- The *procedure pointer binding context*. This is the binding context used by expressions that expect to return a pointer to a procedure.
- The *conditional compilation binding context*. This is the binding context used by expressions within conditional compilation statements.

Unless otherwise specified, expressions use the default binding context to perform name lookup.

5.6.5 Literal Expressions

A *literal expression* consists of a literal.

Static semantics. A literal expression is classified as a value. The declared type of a literal expression is that of the specified token.

```
literal-expression = INTEGER / FLOAT / DATE / STRING / (literal-identifier [type-suffix])
```

Runtime semantics. A literal expression evaluates to the data value represented by the specified token. The value type of a literal expression is that of the specified token.

Any <type-suffix> following a <literal-identifier> has no effect.

5.6.6 Parenthesized Expressions

A *parenthesized expression* consists of an expression enclosed in parentheses.

Static semantics. A parenthesized expression is classified as a value expression, and the enclosed expression MUST able to be evaluated to a simple data value. The declared type of a parenthesized expression is that of the enclosed expression.

```
parenthesized-expression = "(" expression ")"
```

Runtime semantics. A parenthesized expression evaluates to the simple data value of its enclosed expression. The value type of a parenthesized expression is that of the enclosed expression.

5.6.7 TypeOf...Is Expressions

A ***TypeOf...Is*** *expression* is used to check whether the value type of a value is compatible with a given type.

```
typeof-is-expression = "typeof" expression "is" type-expression
```

Static semantics. A **TypeOf...Is** expression is classified as a value and has a declared type of **Boolean**. <expression> MUST be classified as a variable, function, property with a visible **Property Get**, or unbound member and MUST have a declared type of a specific UDT, a specific class, **Object** or **Variant**.

Runtime semantics. The expression evaluates to **True** if any of the following are true:

- The value type of <expression> is the exact type specified by <type-expression>.
- The value type of <expression> is a specific class that implements the interface type specified by <type-expression>.
- The value type of <expression> is any class and <type-expression> specifies the type **Object**.

Otherwise the expression evaluates to **False**.

If the value type of <expression> is **Nothing**, runtime error 91 (Object variable or **With** block variable not set) is raised.

5.6.8 New Expressions

A ***New*** *expression* is used to instantiate an object of a specific class.

```
new-expression = "New" type-expression
```

Static semantics. A **New** expression is invalid if the type referenced by <type-expression> is not instantiable.

A **New** expression is classified as a value and its declared type is the type referenced by <type-expression>.

Runtime semantics. Evaluation of a **New** expression instantiates a new object of the type referenced by <type-expression> and returns that object.

5.6.9 Operator Expressions

There are two kinds of operators. *Unary operators* take one operand and use prefix notation (for example, –x). *Binary operators* take two operands and use infix notation (for example, x + y). With the exception of the relational operators, which always result in **Boolean**, an operator defined for a particular type results in that type. The operands to an operator MUST always be classified as a value; the result of an operator expression is classified as a value.

```
operator-expression = arithmetic-operator-expression / concatenation-operator-expression /
relational-operator-expression / like-operator-expression / is-operator-expression / logical-
operator-expression
```

Static semantics. An operator expression is classified as a value.

5.6.9.1 Operator Precedence and Associativity

When an expression contains multiple binary operators, the *precedence* of the operators controls the order in which the individual binary operators are evaluated. For example, in the expression x + y * z is evaluated as x + (y * z) because the ***** operator has higher precedence than the **+** operator. The following table lists the binary operators in descending order of precedence:

Category	Operators
Primary	All expressions not explicitly listed in this table
Exponentiation	**^**
Unary negation	**-**
Multiplicative	*****, **/**
Integer division	****
Modulus	**Mod**
Additive	**+**, **-**
Concatenation	**&**
Relational	**=**, **<>**, **<**, **>**, **<=**, **>=**, **Like**, **Is**
Logical NOT	**Not**
Logical AND	**And**
Logical OR	**Or**
Logical XOR	**Xor**
Logical EQV	**Eqv**
Logical IMP	**Imp**

When an expression contains two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed. All binary operators are left-

associative, meaning that operations are performed from left to right. Precedence and associativity can be controlled using parenthetical expressions.

5.6.9.2 Simple Data Operators

Simple data operators are operators that first evaluate their operands as simple data values. Specific operators defined in later sections can be designated as simple data operators.

Static semantics. A simple data operator is valid only if it is statically valid to evaluate each of its operands as a simple data value. The declared types of the operands after this static validation are used when determining the declared type of the operator, as defined in each operator's specific section.

Runtime semantics. A simple data operator's operands are first evaluated as simple data values before proceeding with the runtime semantics of operator evaluation.

5.6.9.3 Arithmetic Operators

Arithmetic operators are simple data operators that perform numerical computations on their operands.

```
arithmetic-operator-expression = unary-minus-operator-expression / addition-operator-
expression / subtraction-operator-expression / multiplication-operator-expression / division-
operator-expression / integer-division-operator-expression / modulo-operator-expression /
exponentiation-operator-expression
```

Static semantics. Arithmetic operators are statically resolved as simple data operators.

An arithmetic operator is invalid if the declared type of any operand is an array or a UDT.

For unary arithmetic operators, unless otherwise specified in the specific operator's section, the operator has the following declared type, based on the declared type of its operand:

Operand Declared Type	Operator Declared Type
Byte	**Byte**
Boolean *or* **Integer**	**Integer**
Long	**Long**
LongLong	**LongLong**
Single	**Single**
Double, **String** *or* **String** * *length*	**Double**
Currency	**Currency**
Date	**Date**
Variant	**Variant**

For binary arithmetic operators, unless otherwise specified in the specific operator's section, the operator has the following declared type, based on the declared type of its operands:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Byte	**Byte**	**Byte**
Boolean *or* **Integer**	**Byte**, **Boolean** *or* **Integer**	**Integer**
Byte, **Boolean** *or* **Integer**	**Boolean** *or* **Integer**	**Integer**
Long	**Byte**, **Boolean**, **Integer** *or* **Long**	**Long**
Byte, **Boolean**, **Integer** *or* **Long**	**Long**	**Long**
LongLong	*Any integral numeric type*	**LongLong**
Any integral numeric type	**LongLong**	**LongLong**
Single	**Byte**, **Boolean**, **Integer** *or* **Single**	**Single**
Byte, **Boolean**, **Integer** *or* **Single**	**Single**	**Single**
Single	**Long** *or* **LongLong**	**Double**
Long *or* **LongLong**	**Single**	**Double**
Double, **String** *or* **String** * *length*	*Any integral or floating-point numeric type,* **String** *or* **String** * *length*	**Double**
Any integral or floating-point numeric type, **String** *or* **String** * *length*	**Double**, **String** *or* **String** * *length*	**Double**
Currency	*Any numeric type,* **String** *or* **String** * *length*	**Currency**
Any numeric type, **String** *or* **String** * *length*	**Currency**	**Currency**
Date	*Any numeric type,* **String**, **String** * *length or* **Date**	**Date**
Any numeric type, **String**, **String** * *length or* **Date**	**Date**	**Date**
Any type except an array or UDT	**Variant**	**Variant**
Variant	*Any type except an array or UDT*	**Variant**

Runtime semantics:

- Arithmetic operators are first evaluated as simple data operators.
- If the value type of any operand is an array, UDT or **Error**, runtime error 13 (Type mismatch) is raised.
- Before evaluating the arithmetic operator, its non-**Null** operands undergo **Let**-coercion to the operator's *effective value type*.
- For unary arithmetic operators, unless otherwise specified, the effective value type is determined as follows, based on the value type of the operand:

Operand Value Type	Effective Value Type
Byte	**Byte**
Boolean *or* **Integer** *or* **Empty**	**Integer**
Long	**Long**
LongLong	**LongLong**
Single	**Single**
Double *or* **String**	**Double**
Currency	**Currency**
Date	**Date** *(however, the operand is* ***Let****-coerced to* ***Double*** *instead)*
Decimal	**Decimal**
Null	**Null**

- For binary arithmetic operators, unless otherwise specified, the effective value type is determined as follows, based on the value types of the operands:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Byte	**Byte** *or* **Empty**	**Byte**
Byte *or* **Empty**	**Byte**	**Byte**
Boolean *or* **Integer**	**Byte**, **Boolean**, **Integer** *or* **Empty**	**Integer**
Byte, **Boolean**, **Integer** *or* **Empty**	**Boolean** *or* **Integer**	**Integer**
Empty	**Empty**	**Integer**
Long	**Byte**, **Boolean**, **Integer**, **Long** *or* **Empty**	**Long**
Byte, **Boolean**, **Integer**, **Long** *or* **Empty**	**Long**	**Long**
LongLong	*Any integral numeric type or* **Empty**	**LongLong**
Any integral numeric type or **Empty**	**LongLong**	**LongLong**
Single	**Byte**, **Boolean**, **Integer**, **Single** *or* **Empty**	**Single**
Byte, **Boolean**, **Integer**, **Single** *or* **Empty**	**Single**	**Single**
Single	**Long** *or* **LongLong**	**Double**
Long *or* **LongLong**	**Single**	**Double**

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Double *or* **String**	*Any integral or floating-point numeric type,* **String** *or* **Empty**	
Any integral or floating-point numeric type, **String** *or* **Empty**	**Double** *or* **String**	
Currency	*Any integral or floating-point numeric type,* **Currency**, **String** *or* **Empty**	**Currency**
Any integral or floating-point numeric type, **Currency**, **String** *or* **Empty**	**Currency**	**Currency**
Date	*Any integral or floating-point numeric type,* **Currency**, **String**, **Date** *or* **Empty**	**Date** *(however, the operands are* ***Let****-coerced to* ***Double*** *instead)*
Any integral or floating-point numeric type, **Currency**, **String**, **Date** *or* **Empty**	**Date**	**Date** *(however, the operands are* ***Let****-coerced to* ***Double*** *instead)*
Decimal	*Any numeric type,* **String**, **Date** *or* **Empty**	**Decimal**
Any numeric type, **String**, **Date** *or* **Empty**	**Decimal**	**Decimal**
Null	*Any numeric type,* **String**, **Date**, **Empty**, *or* **Null**	**Null**
Any numeric type, **String**, **Date**, **Empty**, *or* **Null**	**Null**	**Null**
Error	**Error**	**Error**
Error	*Any type except* **Error**	*Runtime error 13 (Type mismatch) is raised.*
Any type except **Error**	**Error**	*Runtime error 13 (Type mismatch) is raised.*

The value type of an arithmetic operator is determined from the value the operator produces, the effective value type and the declared type of its operands as follows:

- If the arithmetic operator produces a value within the valid range of its effective value type, the operator's value type is its effective value type.
- Otherwise, if the arithmetic operator produces a value outside the valid range of its effective value type, arithmetic overflow occurs. The behavior of arithmetic overflow depends on the declared types of the operands:
  - If neither operand has a declared type of Variant, runtime error 6 (Overflow) is raised.
  - If one or both operands have a declared type of **Variant**:

- If the operator's effective value type is **Integer**, **Long**, **Single** or **Double**, the operator's value type is the narrowest type of either **Integer**, **Long** or **Double** such that the operator value is within the valid range of the type**.** If the result does not fit within **Double**, runtime error 6 (Overflow) is raised.
- If the operator's effective value type is **LongLong**, runtime error 6 (Overflow) is raised.
- If the operator's effective value type is **Date**, the operator's value type is **Double**. If the result does not fit within **Double**, runtime error 6 (Overflow) is raised.
- If the operator's effective value type is **Currency** or **Decimal**, runtime error 6 (Overflow) is raised.

The operator's result value is **Let**-coerced to this value type.

Arithmetic operators with an effective value type of **Single** or **Double** perform multiplication, floatingpoint division and exponentiation according to the rules of IEEE 754 arithmetic, which can operate on or result in special values such as positive infinity, negative infinity, positive zero, negative zero or NaN (not a number).

An implementation can choose to perform floating point operations with a higher-precision than the effective value type (such as an "extended" or "long double" type) and coerce the resulting value to the destination declared type. This can be done for performance reasons as some processors are only able to reduce the precision of their floating-point calculations at a severe performance cost.

5.6.9.3.1 Unary - Operator

The *unary* **-** *operator* returns the value of subtracting its operand from 0.

```
unary-minus-operator-expression = "-" expression
```

Static semantics:

- A unary - operator expression has the standard static semantics for unary arithmetic operators.
- A unary - operator expression has the standard static semantics for unary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's declared type:

Operand Declared Type	Operator Declared Type
Byte	**Integer**

Runtime semantics:

- A unary - operator expression has the standard runtime semantics for unary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's effective value type:

Operand Value Type	Effective Value Type
Byte	**Integer**

- The semantics of the unary - operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Byte, **Integer**, **Long**, **LongLong**, **Single**, **Double**, **Currency** or **Decimal**	The result is the operand subtracted from 0.
Date	The **Double** value is the operand subtracted from 0. The result is the **Double** value **Let**-coerced to **Date**. If overflow occurs during the coercion to **Date**, and the operand has a declared type of **Variant**, the result is the **Double** value.
Null	The result is the value **Null**.

5.6.9.3.2 + Operator

The **+** *operator* returns the sum or concatenation of its two operands, depending on their value types.

```
addition-operator-expression = expression "+" expression
```

Static semantics:

- A + operator expression has the standard static semantics for binary arithmetic operators with the following exceptions when determining the operator's declared type:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
String *or* **String** ***** *length*	**String** *or* **String** ***** *length*	**String**

Runtime semantics:

- A + operator expression has the standard runtime semantics for binary arithmetic operators with the following exceptions when determining the operator's effective value type:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
String	**String**	**String**

- The semantics of the + operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Byte, **Integer**, **Long**, **LongLong**, **Single**, **Double**, **Currency** or **Decimal**	The result is the right operand added to the left operand.

Effective Value Type	Runtime Semantics
Date	The **Double** sum is the right operand added to the left operand. The result is the **Double** sum **Let**-coerced to **Date**. If overflow occurs during the coercion to **Date**, and one or both operands have a declared type of **Variant**, the result is the **Double** sum.
String	The result is the right operand string concatenated to the left operand string.
Null	The result is the value **Null**.

5.6.9.3.3 Binary - Operator

The *binary* **-** *operator* (**Unicode** U+2212) returns the difference between its two operands.

```
subtraction-operator-expression = expression "-" expression
```

Static semantics:

- A binary - operator expression has the standard static semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's declared type:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Date	**Date**	**Double**

Runtime semantics:

- A - operator expression has the standard runtime semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's effective value type:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Date	**Date**	**Double**

- The semantics of the - operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Byte, **Integer**, **Long**, **LongLong**, **Single**, **Double**, **Currency** or **Decimal**	The result is the right operand subtracted from the left operand.

Effective Value Type	Runtime Semantics
Date	The **Double** difference is the right operand subtracted from the left operand. The result is the **Double** difference **Let**-coerced to **Date**. If overflow occurs during the coercion to **Date**, and one or both operands have a declared type of **Variant**, the result is the **Double** difference.
Null	The result is the value **Null**.

5.6.9.3.4 * Operator

The ***** *operator* returns the product of its two operands.

```
multiplication-operator-expression = expression "*" expression
```

Static semantics:

- A * operator expression has the standard static semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's declared type:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Currency	**Single**, **Double**, **String** *or* **String** * *length*	**Double**
Single, **Double**, **String** *or* **String** * *length*	**Currency**	**Double**
Date	*Any numeric type*, **String**, **String** * *length or* **Date**	**Double**
Any numeric type, **String**, **String** * *length or* **Date**	**Date**	**Double**

Runtime semantics:

- A * operator expression has the standard runtime semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's effective value type:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Currency	**Single**, **Double** *or* **String**	**Double**
Single, **Double** *or* **String**	**Currency**	**Double**
Date	*Any integral or floating-point numeric type*, **Currency**, **String**, **Date** *or* **Empty**	

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Any integral or floating-point numeric type, **Currency**, **String**, **Date** *or* **Empty**	**Date**	

- The semantics of the * operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Byte, **Integer**, **Long**, **LongLong**, **Currency** *or* **Decimal**	The result is the left operand multiplied with the right operand.
Single *or* **Double**	The result is the left operand multiplied with the right operand. If this results in multiplying positive or negative infinity by 0, runtime error 6 (Overflow) is raised. In this case, if this expression was within the right-hand side of a **Let** assignment and both operands have a declared type of **Double**, the resulting IEEE 754 **Double** special value (such as positive/negative infinity or NaN) is assigned before raising the runtime error.
Null	The result is the value **Null**.

5.6.9.3.5 / Operator

The **/** *operator* returns the quotient of its two operands.

```
division-operator-expression = expression "/" expression
```

Static semantics:

- A / operator expression has the standard static semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's declared type:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Byte, **Boolean**, **Integer**, **Long** *or* **LongLong**	**Byte**, **Boolean**, **Integer**, **Long** *or* **LongLong**	**Double**
Double, **String**, **String *** *length,* **Currency** *or* **Date**	*Any numeric type,* **String**, **String *** *length or* **Date**	**Double**
Any numeric type, **String**, **String *** *length or* **Date**	**Double**, **String**, **String *** *length,* **Currency** *or* **Date**	**Double**

Runtime semantics:

- A / operator expression has the standard runtime semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's effective value type:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Byte, **Boolean**, **Integer**, **Long**, **LongLong** *or* **Empty**	**Byte**, **Boolean**, **Integer**, **Long**, **LongLong** *or* **Empty**	**Double**
Double, **String**, **Currency** *or* **Date**	*Any numeric type*, **String**, **Date** *or* **Empty**	**Double**
Any numeric type, **String**, **Date** *or* **Empty**	**Double**, **String**, **Currency** *or* **Date**	**Double**

- The semantics of the / operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Decimal	The result is the left operand divided by the right operand. If this results in dividing by 0, runtime error 11 (Division by zero) is raised.
Single *or* **Double**	The result is the left operand divided by the right operand. If this results in dividing a nonzero value by 0, runtime error 11 (Division by zero) is raised. If this results in dividing 0 by 0, runtime error 6 (Overflow) is raised, unless the original value type of the left operand is **Single**, **Double**, **String**, or **Date**, and the right operand is **Empty**, in which case runtime error 11 (Division by zero) is raised. In either of these cases, if this expression was within the right-hand side of a **Let** assignment and both operands have a declared type of **Double**, the resulting IEEE 754 **Double** special value (such as positive/negative infinity or NaN) is assigned before raising the runtime error.
Null	The result is the value **Null**.

5.6.9.3.6 \ Operator and Mod Operator

The **** *operator* calculates an integral quotient of its two operands, rounding the quotient towards zero.

The ***Mod*** *operator* calculates the remainder formed when dividing its two operands.

```
integer-division-operator-expression = expression "\" expression
```

```
modulo-operator-expression = expression "mod" expression
```

Static semantics:

- A \ operator expression or **Mod** operator expression has the standard static semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's declared type:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Any floating-point or fixed-point numeric type, **String**, **String *** *length* or **Date**	*Any numeric type*, **String**, **String *** *length* or **Date**	**Long**
Any numeric type, **String**, **String *** *length* or **Date**	*Any floating-point or fixed-point numeric type*, **String**, **String *** *length* or **Date**	**Long**

Runtime semantics:

- A \ operator expression or **Mod** operator expression has the standard runtime semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's effective value type:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Byte	**Empty**	**Integer**
Empty	**Byte**	**Integer**
Boolean *or* **Integer**	**Single**, **Double**, **String**, **Currency**, **Date** *or* **Decimal**	**Integer**
Any floating-point or fixed-point numeric type, **String**, *or* **Date**	*Any numeric type except* **LongLong**, **String**, **Date** *or* **Empty**	**Long**
Any numeric type except **LongLong**, **String**, **Date** *or* **Empty**	*Any floating-point or fixed-point numeric type*, **String**, *or* **Date**	**Long**
LongLong	*Any numeric type*, **String**, **Date** *or* **Empty**	**LongLong**
Any numeric type, **String**, **Date** *or* **Empty**	**LongLong**	**LongLong**

- The semantics of the \ operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Byte, **Integer**, **Long** *or* **LongLong**	The quotient is the left operand divided by the right operand. If the quotient is an integer, the result is the quotient. Otherwise, if the quotient is not an integer, the result is the integer nearest to the quotient that is closer to zero than the quotient. If this results in dividing by 0, runtime error 11 (Division by zero) is raised.
Null	The result is the value **Null**.

- The semantics of the **Mod** operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Byte, **Integer**, **Long** *or* **LongLong**	The quotient is the left operand divided by the right operand. If the quotient is an integer, the result is 0. Otherwise, if the quotient is not an integer, the truncated quotient is the integer nearest to the quotient that is closer to zero than the quotient. The result is the absolute value of the difference between the left operand and the product of the truncated quotient and the right operand. If this results in dividing by 0, runtime error 11 (Division by zero) is raised.
Null	The result is the value **Null**.

5.6.9.3.7 ^ Operator

The **^** *operator* calculates the value of its left operand raised to the power of its right operand.

```
exponentiation-operator-expression = expression "^" expression
```

Static semantics:

- A ^ operator expression has the standard static semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's declared type:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Any numeric type, **String**, **String *** *length or* **Date**	*Any numeric type,* **String**, **String *** *length or* **Date**	**Double**

Runtime semantics:

- A ^ operator expression has the standard runtime semantics for binary arithmetic operators (section 5.6.9.3) with the following exceptions when determining the operator's effective value type:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Any numeric type, **String**, **Date** *or* **Empty**	*Any numeric type,* **String**, **Date** *or* **Empty**	**Double**

- The semantics of the ^ operator depend on the operator's effective value type:

Effective Value Type	Runtime Semantics
Double	The result is the left operand raised to the power of the right operand. If the left operand is 0 and the right operand is 0, the result is 1. If the left operand is 0 and the right operand is negative, runtime error 5 (Invalid procedure call or argument) is raised.
Null	The result is the value **Null**.

5.6.9.4 & Operator

The **&** *operator* is a simple data operator that performs concatenation on its operands. This operator can be used to force concatenation when **+** would otherwise perform addition.

```
concatenation-operator-expression = expression "&" expression
```

Static semantics:

- The & operator is statically resolved as a simple data operator.
- The & operator is invalid if the declared type of either operand is an array or UDT.
- The & operator has the following declared type, based on the declared types of its operands:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Any numeric type, **String**, **String *** *length,* **Date** *or* **Null**	*Any numeric type,* **String**, **String *** *length or* **Date**	**String**
Any numeric type, **String**, **String**	*Any numeric type,* **String**, **String**	**String**
***** *length or* **Date**	***** *length*, **Date** *or* **Null**	
Any type except an array or UDT	**Variant**	**Variant**
Variant	*Any type except an array or UDT*	**Variant**

Runtime semantics:

- The & operator is first evaluated as a simple data operator.
- If the value type of any operand is a non-**Byte** array, UDT or **Error**, runtime error 13 (Type mismatch) is raised.
- Before evaluating the & operator, its non-**Null** operands undergo **Let**-coercion to the operator's value type.
- The operator's value type is determined as follows, based on the value types of the operands:

Left Operand Value Type	Right Operand Value Type	Value Type
Any numeric type, **String**, **Byte()**, **Date**, **Null** *or* **Empty**	*Any numeric type,* **String**, **Byte()**, **Date** *or* **Empty**	**String**
Any numeric type, **String**, **Byte()**, **Date** *or* **Empty**	*Any numeric type,* **String**, **Byte()**, **Date**, **Null** *or* **Empty**	**String**
Null	**Null**	**Null**

- The semantics of the & operator depend on the operator's value type:

Value Type	Runtime Semantics
String	The result is the right operand string concatenated to the left operand string.
Null	The result is the value **Null**.

5.6.9.5 Relational Operators

Relational operators are simple data operators that perform comparisons between their operands.

```
relational-operator-expression = equality-operator-expression / inequality-operator-
expression / less-than-operator-expression / greater-than-operator-expression / less-than-
equal-operator-expression / greater-than-equal-operator-expression
```

Static semantics:

- Relational operators are statically resolved as simple data operators.
- A relational operator is invalid if the declared type of any operand is an array or UDT.
- A relational operator has the following declared type, based on the declared type of its operands:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Any type except an array, UDT or **Variant**	*Any type except an array, UDT or* **Variant**	**Boolean**
Any type except an array or UDT	**Variant**	**Variant**
Variant	*Any type except an array or UDT*	**Variant**

Runtime semantics:

- Relational operators are first evaluated as simple data operators.
- If the value type of any operand is an array or UDT, runtime error 13 (Type mismatch) is raised.
- Before evaluating the relational operator, its non-**Null** operands undergo **Let**-coercion to the operator's *effective value type*.
- The effective value type is determined as follows, based on the value types of the operands:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Byte	**Byte**, **String** *or* **Empty**	**Byte**
Byte, **String** *or* **Empty**	**Byte**	**Byte**
Boolean	**Boolean** *or* **String**	**Boolean**
Boolean *or* **String**	**Boolean**	**Boolean**
Integer	**Byte**, **Boolean**, **Integer**, **String** *or* **Empty**	**Integer**
Byte, **Boolean**, **Integer**, **String** *or* **Empty**	**Integer**	**Integer**
Boolean	**Byte** *or* **Empty**	**Integer**
Byte *or* **Empty**	**Boolean**	**Integer**
Empty	**Empty**	**Integer**
Long	**Byte**, **Boolean**, **Integer**, **Long**, **String** *or* **Empty**	**Long**
Byte, **Boolean**, **Integer**, **Long**, **String** *or* **Empty**	**Long**	**Long**

Left Operand Value Type	Right Operand Value Type	Effective Value Type
LongLong	*Any integral numeric type*, **String** *or* **Empty**	**LongLong**
Any integral numeric type, **String** *or* **Empty**	**LongLong**	**LongLong**
Single	**Byte**, **Boolean**, **Integer**, **Single**, **Double**, **String** *or* **Empty**	**Single**
Byte, **Boolean**, **Integer**, **Single**, **Double**, **String** *or* **Empty**	**Single**	**Single**
Single	**Long**	**Double**
Long	**Single**	**Double**
Double	*Any integral numeric type*, **Double**, **String** *or* **Empty**	
Any integral numeric type, **Double**, **String** *or* **Empty**	**Double**	
String	**String** *or* **Empty**	**String**
String *or* **Empty**	**String**	**String**
Currency	*Any integral or floating-point numeric type*, **Currency**, **String** *or* **Empty**	**Currency**
Any integral or floating-point numeric type, **Currency**, **String** *or* **Empty**	**Currency**	**Currency**
Date	*Any integral or floating-point numeric type*, **Currency**, **String**, **Date** *or* **Empty**	**Date**
Any integral or floating-point numeric type, **Currency**, **String**, **Date** *or* **Empty**	**Date**	**Date**
Decimal	*Any numeric type*, **String**, **Date** *or* **Empty**	**Decimal**
Any numeric type, **String**, **Date** *or* **Empty**	**Decimal**	**Decimal**
Null	*Any numeric type*, **String**, **Date**, **Empty**, *or* **Null**	**Null**
Any numeric type, **String**, **Date**, **Empty**, *or* **Null**	**Null**	**Null**
Error	**Error**	**Error**
Error	*Any type except* **Error**	*Runtime error 13 (Type mismatch) is raised.*

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Any type except **Error**	**Error**	*Runtime error 13 (Type mismatch) is raised.*

- Relational comparisons can test whether operands are considered equal or if one operand is considered less than or greater than the other operand. Such comparisons are governed by the following rules, based on the effective value type:

Effective Value Type	Runtime Semantics
Byte, **Integer**, **Long**, **LongLong**, **Currency**, **Decimal**	The numeric values of the operands are compared. Operands MUST match exactly to be considered equal.
Single or **Double**	The floating-point values of the operands are compared according to the rules of IEEE 754 arithmetic. If either operand is the special value NaN, runtime error 6 (Overflow) is raised.
Boolean	The Boolean values are compared. **True** is considered less than **False**.
String	The String values are compared according to the **Option Compare** *comparison mode (section 5.2.1.1)* setting of the enclosing *module* as follows: ▪ If the active **Option Compare** *comparison mode* is *binary-compare-mode (section 5.2.1.1)*, each byte of the implementation-specific representation of the string data is compared, starting from the byte representing the first character of each string. At any point, if one point is not equal to the other byte, the result of comparing those bytes is the overall result of the comparison. If all bytes in one string are equal to their respective bytes in the other string, but the other string is longer, the longer string is considered greater. Otherwise, if the strings are identical, they are considered equal. ▪ If the active **Option Compare** *comparison mode* is *text-compare-mode (section 5.2.1.1)*, the text of the strings is compared in a case-insensitive manner according to the platform's host-defined *regional settings* for text collation.
Null	The result is the value **Null**.
Error	If both **Error** values are standard error codes, their numeric values (between 0 are 65535) are compared. If either value is an implementation-defined error value, the result of the comparison is undefined.

- There is an exception to the rules in the preceding table when both operands have a declared type of **Variant**, with one operand originally having a value type of **String**, and the other operand originally having a numeric value type. In this case, the numeric operand is considered to be less than (and not equal to) the **String** operand, regardless of their values.

5.6.9.5.1 = Operator

The = *operator* performs a value equality comparison on its operands.

```
equality-operator-expression = expression "=" expression
```

Runtime semantics:

- If the operands are considered equal, **True** is returned. Otherwise, **False** is returned.

5.6.9.5.2 <> Operator

The **<>** *operator* performs a value inequality comparison on its operands. An equivalent alternate operator **><** is also accepted.

```
inequality-operator-expression = expression ( "<"">" / ">""<" ) expression
```

Runtime semantics:

- If the operands are considered not equal, **True** is returned. Otherwise, **False** is returned.

5.6.9.5.3 < Operator

The **<** *operator* performs a less-than comparison on its operands.

```
less-than-operator-expression = expression "<" expression
```

Runtime semantics:

- If the left operand is considered less than the right operand, **True** is returned. Otherwise, **False** is returned.

5.6.9.5.4 > Operator

The **>** *operator* performs a greater-than comparison on its operands.

```
greater-than-operator-expression = expression ">" expression
```

Runtime semantics:

- If the left operand is considered greater than the right operand, **True** is returned. Otherwise, **False** is returned.

5.6.9.5.5 <= Operator

The **<=** *operator* performs a less-than-or-equal comparison on its operands.

```
less-than-equal-operator-expression = expression ( "<""=" / "=""<" ) expression
```

Runtime semantics:

- If the left operand is considered less than or equal to the right operand, **True** is returned. Otherwise, **False** is returned.

5.6.9.5.6 >= Operator

The **>=** *operator* performs a greater-than-or-equal comparison on its operands.

```
greater-than-equal-operator-expression = expression ( ">""=" / "="">" ) expression
```

Runtime semantics:

- If the left operand is considered greater than or equal to the right operand, **True** is returned. Otherwise, **False** is returned.

5.6.9.6 Like Operator

The ***Like*** *operator* is a simple data operator that performs a string matching test of the source string in the left operand against the pattern string in the right operand.

```
like-operator-expression = expression "like" like-pattern-expression
like-pattern-expression = expression
```

Static semantics:

- The **Like** operator is statically resolved as a simple data operator.
- A **Like** operator expression is invalid if the declared type of any operand is an array or a UDT.
- A **Like** operator has the following declared type, based on the declared type of its operands:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Any type except an array, UDT or **Variant**	*Any type except an array, UDT or* **Variant**	**Boolean**
Any type except an array or UDT	**Variant**	**Variant**
Variant	*Any type except an array or UDT*	**Variant**

Runtime semantics:

- The **Like** operator is first evaluated as a simple data operator.
- If either <expression> or <like-pattern-expression> is Null, the result is Null.
- Otherwise, <expression> and <like-pattern-expression> are both Let-coerced to String. The grammar for the String value of <like-pattern-expression> is interpreted as <like-pattern-string>, according to the following grammar:

```
like-pattern-string = *like-pattern-element
like-pattern-element = like-pattern-char / "?" / "#" / "*" / like-pattern-charlist
like-pattern-char = <Any character except "?", "#", "*" and "[" >
like-pattern-charlist = "[" ["!"] ["-"] *like-pattern-charlist-element ["-"] "]"
like-pattern-charlist-element = like-pattern-charlist-char / like-pattern-charlist-range
like-pattern-charlist-range = like-pattern-charlist-char "-" like-pattern-charlist-char
like-pattern-charlist-char = <Any character except "-" and "]">
```

- The pattern in <like-pattern-expression> is matched one <like-pattern-element> at a time to the characters in <expression> until either:
  - All characters of <expression> and <like-pattern-expression> have been matched. In this case, the result is **True**.
  - Either <expression> or <like-pattern-expression> is fully matched, while the other string still has unmatched characters. In this case, the result is **False**.
  - A <like-pattern-element> does not match the next characters in <expression>. In this case, the result is **False**.
  - The next characters in <like-pattern-expression> do not form a valid, complete <like-pattern-element> according to the grammar. In this case, runtime error 93 (Invalid pattern string) is raised. Note that this runtime error is only raised if no other result has been produced before pattern matching proceeds far enough to encounter this error in the pattern.
- String matching uses the **Option Compare** *comparison mode (section 5.2.1.1)* setting of the enclosing *module*, as well as any implementation-defined *regional settings* related to text collation. When the *comparison mode* is *text-compare-mode (section 5.2.1.1)*, some number of actual characters in <expression> can match a different number of characters in the pattern, according to the host-defined regional text collation settings. This means that the single pattern character "æ" can match the expression characters "ae". A pattern character can also match just part of an expression character, such as the two pattern characters "ae" each matching part of the single expression character "æ".
- Each <like-pattern-element> in the pattern has the following meaning:

Pattern element	Meaning
<like-pattern-char>	Matches the specified character.
?	Matches any single actual character in the expression, or the rest of a partially matched actual character. When the *comparison mode* is *text-compare-mode*, the **?** pattern element matches all the way to the end of one actual character in <expression>, which can be just the last part of a partially matched expression character. This means that the expression "æ" can be matched by the pattern "a?", but might not be matched by the pattern "?e".
#	Matches a single character representing a digit.
*****	Matches zero or more characters. When a ***** pattern element is encountered, the rest of the pattern is immediately checked to ensure it can form a sequence of valid, complete <like-pattern-element> instances according to the grammar. If this is not possible, runtime error 93 (Invalid pattern string) is raised.

Pattern element	Meaning
	When the *comparison mode* is *text-compare-mode*, the ***** pattern element can match part of a character. This means that the expression "æ" can be matched by the pattern "a*" or the pattern "*e".
<like-pattern-charlist>	Matches one of the characters in the specified character list. A <like-pattern-charlist> contains a sequence of <like-pattern-charlist-element> instances, representing the set of possible characters that can be matched. Each <like-pattern-charlist-element> can be one of the following: ▪ <like-pattern-charlist-char>: This adds the specified character to the character list. ▪ <like-pattern-charlist-range>: This adds a range of characters to the character list, including all characters considered greater than or equal to the first <like-pattern-charlist-char> and considered less than or equal to the second <like-pattern-charlist-char>. If the end character of this range is considered less than the start character, runtime error 93 (Invalid pattern string) is raised. Semantics are undefined if a compound character such as "æ" that can match multiple expression characters is used within a <like-pattern-charlist-range> when the *comparison mode* is *text-compare-mode*. If the optional "-" is specified at the beginning or end of <like-pattern-charlist>, the character "-" is included in the character list. If the optional "!" is specified at the beginning of <like-pattern-charlist>, this pattern element will instead match characters not in the specified character list. When the *comparison mode* is *text-compare-mode*, the first specified element of the character list that can match part of the actual expression character is chosen as the match. This means that the expression "æ" can be matched by the pattern "a[ef]" or "[æa]", but might not be matched by the pattern "[aæ]".

5.6.9.7 Is Operator

The ***Is*** *operator* performs reference equality comparison.

```
is-operator-expression = expression "is" expression
```

Static semantics:

- Each expression MUST be classified as a value and the declared type of each expression MUST be a specific class, **Object** or **Variant**.
- An **Is** operator has a declared type of **Boolean**.

Runtime semantics:

- The expression evaluates to **True** if both values refer to the same instance or **False** otherwise.
- If either expression has a value type other than a specific class or **Nothing**, runtime error 424 (Object required) is raised.

5.6.9.8 Logical Operators

Logical operators are simple data operators that perform bitwise computations on their operands.

```
logical-operator-expression = not-operator-expression / and-operator-expression / or-
operator-expression / xor-operator-expression / imp-operator-expression / eqv-operator-
expression
```

Static semantics:

- Logical operators are statically resolved as simple data operators.
- A logical operator is invalid if the declared type of any operand is an array or a UDT.
- For unary logical operators, the operator has the following declared type, based on the declared type of its operand:

Operand Declared Type	Operator Declared Type
Byte	**Byte**
Boolean	**Boolean**
Integer	**Integer**
Any floating-point or fixed-point numeric type, **Long***,* **String***,* **String** ***** *length or* **Date**	**Long**
LongLong	**LongLong**
Variant	**Variant**

- For binary logical operators, the operator has the following declared type, based on the declared type of its operands:

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Byte	**Byte**	**Byte**
Boolean	**Boolean**	**Boolean**
Byte *or* **Integer**	**Boolean** *or* **Integer**	**Integer**
Boolean *or* **Integer**	**Byte** *or* **Integer**	**Integer**

Left Operand Declared Type	Right Operand Declared Type	Operator Declared Type
Any floating-point or fixed-point numeric type, **Long**, **String**, **String** ***** *length or* **Date**	*Any numeric type except* **LongLong**, **String**, **String** ***** *length or* **Date**	**Long**
Any numeric type except **LongLong**, **String**, **String** ***** *length or* **Date**	*Any floating-point or fixed-point numeric type,* **Long**, **String**, **String** ***** *length or* **Date**	**Long**
LongLong	*Any numeric type,* **String**, **String** ***** *length or* **Date**	**LongLong**
Any numeric type, **String**, **String** ***** *length or* **Date**	**LongLong**	**LongLong**
Any type except an array or UDT	**Variant**	**Variant**

Runtime semantics:

- Logical operators are first evaluated as simple data operators.
- If the value type of any operand is an array, UDT or **Error**, runtime error 13 (Type mismatch) is raised.
- Before evaluating the logical operator, its non-**Null** operands undergo **Let**-coercion to the operator's *effective value type*.
- For unary logical operators, the effective value type is determined as follows, based on the value type of the operand:

Operand Value Type	Effective Value Type
Byte	**Byte**
Boolean *or* **Integer** *or* **Empty**	**Integer**
Long	**Long**
LongLong	**LongLong**
Single	**Single**
Double *or* **String**	**Double**
Currency	**Currency**
Date	**Date** *(however, the operand is* ***Let****-coerced to* ***Double*** *instead)*
Decimal	**Decimal**
Null	**Null**

- For binary logical operators, if either operator is null, the effective value type is determined as follows, based on the value types of the operands:

Left Operand Value Type	Right Operand Value Type	Effective Value Type
Byte *or* **Null**	**Byte**	**Byte**
Byte	**Byte** *or* **Null**	**Byte**
Boolean *or* **Null**	**Boolean**	**Boolean** *(however, the operands are **Let**-coerced to **Integer** instead)*
Boolean	**Boolean** *or* **Null**	**Boolean** *(however, the operands are **Let**-coerced to **Integer** instead)*
Byte, **Boolean**, **Integer**, **Null** *or* **Empty**	**Integer** *or* **Empty**	**Integer**
Integer *or* **Empty**	**Byte**, **Boolean**, **Integer**, **Null** *or* **Empty**	**Integer**
Byte	**Boolean**	**Integer**
Boolean	**Byte**	**Integer**
Any floating-point or fixed-point numeric type, **Long**, **String**, **Date** *or* **Empty**	*Any numeric type except* **LongLong**, **String**, **Date**, **Null** *or* **Empty**	**Long**
Any numeric type except **LongLong**, **String**, **Date**, **Null** *or* **Empty**	*Any floating-point or fixed-point numeric type*, **Long**, **String**, **Date** *or* **Empty**	**Long**
LongLong	*Any numeric type*, **String**, **Date** *or* **Empty**	**LongLong**
Any numeric type, **String**, **Date** *or* **Empty**	**LongLong**	**LongLong**
Null	**Null**	**Null**

- The value type of a logical operator is determined from the value the operator produces:
  - If the logical operator produces a value other than Null, the operator's value type is its effective value type. The operator's result value is Let-coerced to this value type.
  - Otherwise, if the logical operator produces **Null**, the operator's value is **Null**.

5.6.9.8.1 Not Operator

The ***Not** operator* performs a bitwise negation on its operand.

```
not-operator-expression = "not" expression
```

Runtime semantics:

- The operation to produce the result is determined based on the values of the operand, as follows:

Operand Value	Result
Integral value	*Bitwise* **Not** *of operand*
Null	**Null**

- If a bitwise **Not** of the operand is indicated, the result is produced by generating a corresponding result bit for each identically positioned bit in the implementation format of the operand according to the following table:

Operand Bit	Result Bit
0	**1**
1	**0**

5.6.9.8.2 And Operator

The ***And*** *operator* performs a bitwise conjunction on its operands.

```
and-operator-expression = expression "and" expression
```

Runtime semantics:

- The operation to produce the result is determined based on the values of the operands, as follows:

Left Operand Value	Right Operand Value	Result
Integral value	*Integral value*	*Bitwise* **And** *of operands*
Integral value other than **0**	**Null**	**Null**
0	**Null**	**0**
Null	*Integral value other than* **0**	**Null**
Null	**0**	**0**
Null	**Null**	**Null**

- If a bitwise **And** of the operands is indicated, the result is produced by generating a corresponding result bit for each pair of identically positioned bits in the implementation format of the operands according to the following table:

Left Operand Bit	Right Operand Bit	Result Bit
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

5.6.9.8.3 Or Operator

The ***Or*** *operator* performs a bitwise disjunction on its operands.

```
or-operator-expression = expression "or" expression
```

Runtime semantics:

- The operation to produce the result is determined based on the values of the operands, as follows:

Left Operand Value	Right Operand Value	Result
Integral value	*Integral value*	*Bitwise* **Or** *of operands*
Integral value	**Null**	*Left operand*
Null	*Integral value*	*Right operand*
Null	**Null**	**Null**

- If a bitwise **Or** of the operands is indicated, the result is produced by generating a corresponding result bit for each pair of identically positioned bits in the implementation format of the operands according to the following table:

Left Operand Bit	Right Operand Bit	Result Bit
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**1**

5.6.9.8.4 Xor Operator

The ***Xor*** *operator* performs a bitwise exclusive disjunction on its operands.

```
xor-operator-expression = expression "xor" expression
```

Runtime semantics:

- The operation to produce the result is determined based on the values of the operands, as follows:

Left Operand Value	Right Operand Value	Result
Integral value	*Integral value*	*Bitwise* **Xor** *of operands*
Integral value	**Null**	**Null**
Null	*Integral value*	**Null**

Left Operand Value	Right Operand Value	Result
Null	**Null**	**Null**

- If a bitwise **Xor** of the operands is indicated, the result is produced by generating a corresponding result bit for each pair of identically positioned bits in the implementation format of the operands according to the following table:

Left Operand Bit	Right Operand Bit	Result Bit
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**
1	**1**	**0**

5.6.9.8.5 Eqv Operator

The ***Eqv*** *operator* performs a bitwise material equivalence on its operands.

```
eqv-operator-expression = expression "eqv" expression
```

Runtime semantics:

- The operation to produce the result is determined based on the values of the operands, as follows:

Left Operand Value	Right Operand Value	Result
Integral value	*Integral value*	*Bitwise* **Eqv** *of operands*
Integral value	**Null**	**Null**
Null	*Integral value*	**Null**
Null	**Null**	**Null**

- If a bitwise **Eqv** of the operands is indicated, the result is produced by generating a corresponding result bit for each pair of identically positioned bits in the implementation format of the operands according to the following table:

Left Operand Bit	Right Operand Bit	Result Bit
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

5.6.9.8.6 Imp Operator

The ***Imp*** *operator* performs a bitwise material implication on its operands.

```
imp-operator-expression = expression "imp" expression
```

Runtime semantics:

- The operation to produce the result is determined based on the values of the operands, as follows:

Left Operand Value	Right Operand Value	Result
Integral value	*Integral value*	*Bitwise* **Imp** *of operands*
-1	**Null**	**Null**
Integral value other than **-1**	**Null**	*Bitwise* **Imp** *of left operand and* **0**
Null	*Integral value other than* **0**	*Right operand*
Null	**0**	**Null**
Null	**Null**	**Null**

- If a bitwise **Imp** of the operands is indicated, the result is produced by generating a corresponding result bit for each pair of identically positioned bits in the implementation format of the operands according to the following table:

Left Operand Bit	Right Operand Bit	Result Bit
0	**0**	**1**
0	**1**	**1**
1	**0**	**0**
1	**1**	**1**

5.6.10 Simple Name Expressions

A *simple name expression* consists of a single identifier with no qualification or argument list.

```
simple-name-expression = name
```

Static semantics. Simple name expressions are resolved and classified by matching <name> against a set of *namespace tiers* in order.

The first tier where the name value of <name> matches the name value of at least one element of the tier is the selected tier. The match that the simple name expression references is chosen as follows:

- If the selected tier contains matches from multiple referenced projects, the matches from the project that has the highest reference precedence are retained and all others are discarded.

- If both an **Enum** type match and an **Enum** member match are found within the selected tier, the match that is defined later in the module is discarded. In the case where an **Enum** member match is defined within the body of an **Enum** type match, the **Enum** member match is considered to be defined later in the module.
- If there is a single match remaining in the selected tier, that match is chosen.
- If there are 2 or more matches remaining in the selected tier, the simple name expression is invalid.

If all tiers have no matches, unless otherwise specified, the simple name expression is invalid.

If <name> specifies a type character, and this type character's associated type does not match the declared type of the match, the simple name expression is invalid.

The simple name expression refers to the chosen match, inheriting the declared type, if any, from the match.

Simple name expressions are classified based on the entity they match:

Match	Simple Name Expression Classification
Constant or **Enum** *member*	Value expression
Variable, including implicitly-defined variables	Variable expression
Property	Property expression
Function	Function expression
Subroutine	Subroutine expression
Project	Project expression
Procedural module	Procedural module expression
Class module, UDT or **Enum** *type*	Type expression

The namespace tiers under the default binding context are as follows, in order of precedence:

- **Procedure namespace:** A local variable, reference parameter binding or constant whose implicit or explicit definition precedes this expression in an enclosing procedure.
- **Enclosing Module namespace:** A variable, constant, **Enum** type, **Enum** member, property, function or subroutine defined at the module-level in the enclosing module.
- **Enclosing Project namespace:** The enclosing project itself, a referenced project, or a procedural module contained in the enclosing project.
- **Other Procedural Module in Enclosing Project namespace:** An accessible variable, constant, **Enum** type, **Enum** member, property, function or subroutine defined in a procedural module within the enclosing project other than the enclosing module.
- **Referenced Project namespace:** An accessible procedural module contained in a referenced project.

- **Module in Referenced Project namespace:** An accessible variable, constant, **Enum** type, **Enum** member, property, function or subroutine defined in a procedural module or as a member of the default instance of a global class module within a referenced project.

There is a special exception to these namespace tiers when the match has the name value "Left":

- If the match has the name value "Left", references a function or subroutine that has no parameters, or a property with a **Property Get** that has no parameters, the declared type of the match is any type except a specific class, **Object** or **Variant**, and this simple name expression is the <l-expression> within an index expression with an argument list containing 2 arguments, discard the match and continue searching for a match on lower tiers.

Under the default binding context, if all tiers have no matches:

- If the variable declaration mode for the enclosing module is explicit-mode, the simple name expression is invalid.
- Otherwise, if the variable declaration mode for the enclosing module is implicit-mode, a new local variable is *implicitly declared* in the current procedure as if by a local variable declaration statement immediately preceding this statement with a <variable-declaration-list> containing a single <variable-dcl> element consisting of the text of <name>. This newly created variable is the match.

The namespace tiers under the type binding context are as follows, in order of precedence:

- **Enclosing Module namespace:** A UDT or **Enum** type defined at the module-level in the enclosing module.
- **Enclosing Project namespace:** The enclosing project itself, a referenced project, or a procedural module or class module contained in the enclosing project.
- **Other Module in Enclosing Project namespace:** An accessible UDT or **Enum** type defined in a procedural module or class module within the enclosing project other than the enclosing module.
- **Referenced Project namespace:** An accessible procedural module or class module contained in a referenced project.
- **Module in Referenced Project namespace:** An accessible UDT or **Enum** type defined in a procedural module or class module within a referenced project.

The namespace tiers under the procedure pointer binding context are as follows, in order of precedence:

- **Enclosing Module namespace:** A function, subroutine or property with a **Property Get** defined at the module-level in the enclosing module.
- **Enclosing Project namespace:** The enclosing project itself or a procedural module contained in the enclosing project.
- **Other Procedural Module in Enclosing Project namespace:** An accessible function, subroutine or property with a **Property Get** defined in a procedural module within the enclosing project other than the enclosing module.

The namespace tiers under the conditional compilation binding context are as follows, in order of precedence:

- **Enclosing Module namespace:** A conditional compilation constant defined at the module-level in the enclosing module.
- **Enclosing Project namespace:** A conditional compilation constant defined in an implementation-defined way by the enclosing project itself.

5.6.11 Instance Expressions

An *instance expression* consists of the keyword **Me**.

```
instance-expression = "me"
```

Static semantics. An instance expression is classified as a value. The declared type of an instance expression is the type defined by the class module containing the enclosing procedure. It is invalid for an instance expression to occur within a procedural module.

Runtime semantics. The keyword **Me** represents the *current instance* of the type defined by the enclosing class module and has this type as its value type.

5.6.12 Member Access Expressions

A *member access expression* is used to reference a *member* of an entity.

```
member-access-expression = l-expression NO-WS "." unrestricted-name
member-access-expression =/ l-expression line-continuation "." unrestricted-name
```

Static semantics. The semantics of a member access expression depend on the binding context.

A member access expression under the default binding context is valid only if one of the following is true:

- <l-expression> is classified as a variable, a property or a function and one of the following is true:
  - The declared type of <l-expression> is a UDT type or specific class, this type has an accessible member named <unrestricted-name>, <unrestricted-name> either does not specify a type character or specifies a type character whose associated type matches the declared type of the member, and one of the following is true:
    - The member is a variable, property or function. In this case, the member access expression is classified as a variable, property or function, respectively, refers to the member, and has the same declared type as the member.
    - The member is a subroutine. In this case, the member access expression is classified as a subroutine and refers to the member.
  - The declared type of <l-expression> is **Object** or **Variant**. In this case, the member access expression is classified as an unbound member and has a declared type of **Variant**.
- <l-expression> is classified as an unbound member. In this case, the member access expression is classified as an unbound member and has a declared type of **Variant**.
- <l-expression> is classified as a project, this project is either the enclosing project or a referenced project, and one of the following is true:

- <l-expression> refers to the enclosing project and <unrestricted-name> is either the name of the enclosing project or a referenced project. In this case, the member access expression is classified as a project and refers to the specified project.
- The project has an accessible procedural module named <unrestricted-name>. In this case, the member access expression is classified as a procedural module and refers to the specified procedural module.
- The project does not have an accessible procedural module named <unrestricted-name> and exactly one of the procedural modules within the project has an accessible member named <unrestricted-name>, <unrestricted-name> either does not specify a type character or specifies a type character whose associated type matches the declared type of the member, and one of the following is true:
  - The member is a variable, property or function. In this case, the member access expression is classified as a variable, property or function, respectively, refers to the member, and has the same declared type as the member.
  - The member is a subroutine. In this case, the member access expression is classified as a subroutine and refers to the member.
  - The member is a value. In this case, the member access expression is classified as a value with the same declared type as the member.

- <l-expression> is classified as a procedural module, this procedural module has an accessible member named <unrestricted-name>, <unrestricted-name> either does not specify a type character or specifies a type character whose associated type matches the declared type of the member, and one of the following is true:
  - The member is a variable, property or function. In this case, the member access expression is classified as a variable, property or function, respectively, and has the same declared type as the member.
  - The member is a subroutine. In this case, the member access expression is classified as a subroutine.
  - The member is a value. In this case, the member access expression is classified as a value with the same declared type as the member.
- <l-expression> is classified as a type, this type is an **Enum** type, and this type has an enum member named <unrestricted-name>. In this case, the member access expression is classified as a value with the same declared type as the enum member.

A member access expression under the type binding context is valid only if one of the following is true:

- <l-expression> is classified as a project, this project is either the enclosing project or a referenced project, and one of the following is true:
  - <l-expression> refers to the enclosing project and <unrestricted-name> is either the name of the enclosing project or a referenced project. In this case, the member access expression is classified as a project and refers to the specified project.
  - The project has an accessible procedural module named <unrestricted-name>. In this case, the member access expression is classified as a procedural module and refers to the specified procedural module.
  - The project has an accessible class module named <unrestricted-name>. In this case, the member access expression is classified as a type and refers to the specified class.

- The project does not have an accessible module named <unrestricted-name> and exactly one of the procedural modules within the project contains a UDT or **Enum** definition named <unrestricted-name>. In this case, the member access expression is classified as a type and refers to the specified UDT or enum.

- <l-expression> is classified as a procedural module or a type referencing a class defined in a class module, and one of the following is true:

  - This module has an accessible UDT or **Enum** definition named <unrestricted-name>. In this case, the member access expression is classified as a type and refers to the specified UDT or **Enum** type.

A member access expression under the procedure pointer binding context is valid only if <l-expression> is classified as a procedural module, this procedural module has an accessible function or subroutine with the same name value as <unrestricted-name>, and <unrestricted-name> either does not specify a type character or specifies a type character whose associated type matches the declared type of the function or subroutine. In this case, the member access expression is classified as a function or subroutine, respectively.

5.6.13 Index Expressions

An *index expression* is used to parameterize an expression by adding an argument list to its *argument list queue*.

```
index-expression = l-expression "(" argument-list ")"
```

Static semantics. An index expression is valid only if under the default binding context and one of the following is true:

- <l-expression> is classified as a variable, or <l-expression> is classified as a property or function with a parameter list that cannot accept any parameters and an <argument-list> that is not empty, and one of the following is true:

  - The declared type of <l-expression> is **Object** or **Variant**, and <argument-list> contains no named arguments. In this case, the index expression is classified as an unbound member with a declared type of **Variant**, referencing <l-expression> with no member name.

  - The declared type of <l-expression> is a specific class, which has a public default **Property Get**, **Property Let**, function or subroutine, and one of the following is true:

    - This default member's parameter list is compatible with <argument-list>. In this case, the index expression references this default member and takes on its classification and declared type.

    - This default member cannot accept any parameters. In this case, the static analysis restarts recursively, as if this default member was specified instead for <l-expression> with the same <argument-list>.

  - The declared type of <l-expression> is an array type, an empty argument list has not already been specified for it, and one of the following is true:

    - <argument-list> represents an empty argument list. In this case, the index expression takes on the classification and declared type of <l-expression> and references the same array.

    - <argument-list> represents an argument list with a number of positional arguments equal to the rank of the array, and with no named arguments. In this case, the index expression

references an individual element of the array, is classified as a variable and has the declared type of the array's element type.

- <l-expression> is classified as a property or function and its parameter list is compatible with <argument-list>. In this case, the index expression references <l-expression> and takes on its classification and declared type.

- <l-expression> is classified as a subroutine and its parameter list is compatible with <argument-list>. In this case, the index expression references <l-expression> and takes on its classification and declared type.

- <l-expression> is classified as an unbound member. In this case, the index expression references <l-expression>, is classified as an unbound member and its declared type is **Variant**.

In any of these cases where the index expression is valid, the resulting expression adopts the argument list queue of <l-expression> as its own, adding <argument-list> to the end of the queue. The argument list queue of <l-expression> is cleared.

5.6.13.1 Argument Lists

An *argument list* represents an ordered list of positional arguments and a set of named arguments that are used to parameterize an expression.

```
argument-list = [positional-or-named-argument-list]
positional-or-named-argument-list = *(positional-argument ",") required-positional-argument
positional-or-named-argument-list =/   *(positional-argument ",") named-argument-list
positional-argument = [argument-expression]
required-positional-argument = argument-expression
named-argument-list = named-argument *("," named-argument)
named-argument = unrestricted-name ":""=" argument-expression
argument-expression = ["byval"] expression
argument-expression =/  addressof-expression
```

Static semantics. An argument list is composed of positional arguments and named arguments.

If <positional-or-named-argument-list> is omitted, the argument list is said to represent an *empty argument list* and has no positional arguments and no named arguments.

Each <positional-argument> or <required-positional-argument> represents a specified *positional argument*. If a specified positional argument omits its <argument-expression>, the specified positional argument is said to be *omitted*. Each specified positional argument consists of a position based on its order in the argument list from left to right, as well as an expression from its <argument-expression>, if not omitted.

Each <named-argument > represents a *named argument*. Each named argument consists of a name value from its <unrestricted-name>, as well as an expression from its <argument-expression>.

The "byval" keyword flags a specific argument as being a ***ByVal** argument*. It is invalid for an argument list to contain a **ByVal** argument unless it is the argument list for an invocation of an external procedure.

5.6.13.2 Argument List Queues

An *argument list queue* is a FIFO (first-in-first-out) sequence of argument lists belonging to a particular expression.

During evaluation and member resolution, argument lists within a queue are statically consumed to determine that an expression is valid. At runtime, these argument lists start out unconsumed and are consumed again as they are applied to specific array or procedure references. An argument list is considered empty, either statically or at runtime, if the queue has no argument lists or if all of its argument lists are currently consumed.

5.6.14 Dictionary Access Expressions

A *dictionary access expression* is an alternate way to invoke an object's *default member* with a **String** parameter.

```
dictionary-access-expression = l-expression  NO-WS "!" NO-WS unrestricted-name
dictionary-access-expression =/  l-expression  line-continuation "!" NO-WS unrestricted-name
dictionary-access-expression =/  l-expression  line-continuation "!" line-continuation
unrestricted-name
```

Static semantics. A dictionary access expression is invalid if the declared type of <l-expression> is a type other than a specific class, **Object** or **Variant**.

A dictionary access expression is syntactically translated into an index expression with the same expression for <l-expression> and an argument list with a single positional argument with a declared type of **String** and a value equal to the name value of <unrestricted-name>.

5.6.15 With Expressions

A ***With*** *expression* is a member access or dictionary access expression with its <l-expression> implicitly supplied by the innermost enclosing **With** block.

```
with-expression = with-member-access-expression / with-dictionary-access-expression

with-member-access-expression = "." unrestricted-name
with-dictionary-access-expression = "!" unrestricted-name
```

Static semantics. A <with-member-access-expression> or <with-dictionary-access-expression> is statically resolved as a normal member access or dictionary access expression, respectively, as if the innermost enclosing **With** block variable was specified for <l-expression>. If there is no enclosing **With** block, the <with-expression> is invalid.

5.6.16 Constrained Expressions

Constrained expressions are special-purpose expressions that statically permit only a subset of the full expression grammar.

5.6.16.1 Constant Expressions

A *constant expression* is an expression usable in contexts which require a value that can be fully evaluated statically.

```
constant-expression = expression
```

Static semantics. A constant expression is valid only when <expression> is composed solely of the following constructs:

- Numeric, **String**, **Date**, **Empty**, **Null**, or **Nothing** literal.
- Reference to a module-level constant.
- Reference to a procedure-level constant explicitly declared in the enclosing procedure, if any.
- Reference to a member of an enumeration type.
- Parenthesized subexpression, provided the subexpression is itself valid as a constant expression.
  ☐ **-** or **Not** unary operator, provided the operand is itself valid as a constant expression.
- **+**, **-**, *****, **^**, **Mod**, **/**, ****, **&**, **And**, **Or**, **Xor**, **Eqv**, **Imp**, **=**, **<**, **>**, **<>**, **<=**, **=>** or **Like** binary operator, provided each operand is itself valid as a constant expression.
- The **Is** binary operator, provided each operand is itself valid as a constant expression.
- Simple name expression invoking the VBA intrinsic function **Int**, **Fix**, **Abs**, **Sgn**, **Len**, **LenB**, **CBool**, **CByte**, **CCur**, **CDate**, **CDbl**, **CInt**, **CLng**, **CLngLng**, **CLngPtr**, **CSng**, **CStr** or **CVar**.

References within constant expressions might not refer to the implicit **With** block variable.

The *constant value* of a constant expression is determined statically by evaluating <expression> as if it was being evaluated at runtime.

5.6.16.2 Conditional Compilation Expressions

A *conditional compilation expression* is an expression usable within conditional compilation statements.

```
cc-expression = expression
```

Static semantics. The semantics of conditional compilation expressions are only defined when <expression> is composed solely of the following constructs:

- Numeric, **String**, **Date**, **Empty**, **Null**, or **Nothing** literal.
- Reference to a conditional compilation constant.
- Parenthesized subexpression, provided the subexpression is itself valid as a conditional compilation expression.
- The **-** and **Not** unary operators, provided the operand is itself valid as a conditional compilation expression.
- The **+**, **-**, *****, **^**, **Mod**, **/**, ****, **&**, **And**, **Or**, **Xor**, **Eqv**, **Imp**, **=**, **<**, **>**, **<>**, **<=**, **=>** or **Like**, provided each operand is itself valid as a conditional compilation expression.
- The **Is** binary operator, provided each operand is itself valid as a conditional compilation expression.
- Simple name expression invoking the VBA intrinsic function **Int**, **Fix**, **Abs**, **Sgn**, **Len**, **LenB**, **CBool**, **CByte**, **CCur**, **CDate**, **CDbl**, **CInt**, **CLng**, **CLngLng**, **CLngPtr**, **CSng**, **CStr** or **CVar**.

References within conditional compilation expressions might not refer to the implicit **With** block variable.

The *constant value* of a conditional compilation expression is determined statically by evaluating <expression> as if it was being evaluated at runtime with conditional compilation constants being replaced by their defined values.

5.6.16.3 Boolean Expressions

```
boolean-expression = expression
```

Static Semantics. A <boolean-expression> is invalid if a **Let** coercion from the declared type of <expression> to **Boolean** is invalid. The declared type of a <boolean-expression> is **Boolean**.

Runtime Semantics.

- If <expression> does not have the data value **Null**, <expression> is **Let**-coerced to **Boolean**, and the value of <expression> is this coerced value.
- Otherwise, if <expression> has the data value **Null**, the value of <expression> is **False**.

5.6.16.4 Integer Expressions

```
integer-expression = expression
```

Static Semantics.

An <integer-expression> is invalid if a **Let** coercion from the declared type of <expression> to **Long** is invalid. The declared type of an <integer-expression> is **Long**.

Runtime Semantics. The value of an <integer-expression> is the value of its <expression> **Let**-coerced to **Long**.

5.6.16.5 Variable Expressions

```
variable-expression = l-expression
```

Static Semantics.

A <variable-expression> is invalid if it is classified as something other than a variable or unbound member.

5.6.16.6 Bound Variable Expressions

```
bound-variable-expression = l-expression
```

Static Semantics.

A <bound-variable-expression> is invalid if it is classified as something other than a variable expression. The expression is invalid even if it is classified as an unbound member expression that could be resolved to a variable expression.

5.6.16.7 Type Expressions

```
type-expression = BUILTIN-TYPE / defined-type-expression
```

```
defined-type-expression = simple-name-expression / member-access-expression
```

Static Semantics. A <defined-type-expression> performs name binding under the type binding context. A <defined-type-expression> is invalid if it is not classified as a type. A <type-expression> is classified as a type.

5.6.16.8 AddressOf Expressions

addressof-expression = "addressof" procedure-pointer-expression

```
procedure-pointer-expression = simple-name-expression / member-access-expression
```

Static semantics.

<procedure-pointer-expression> performs name binding under the procedure pointer binding context, and MUST be classified as a subroutine, function or a property with a **Property Get**. The procedure referenced by this expression is the *referenced procedure*.

An ***AddressOf*** *expression* is invalid if <procedure-pointer-expression> refers to a subroutine, function or property defined in a class module and the expression is qualified with the name of the class module.

The **AddressOf** expression is classified as a value expression. The declared type and value type of an **AddressOf** expression is implementation-defined, and can be **Long**, **LongLong** or other implementation-defined types.

Runtime semantics. The result is an implementation-defined value capable of serving as an invocable reference to the referenced procedure when passed directly as a parameter to an external procedure call. An implementation where such a value would exceed the range of the integral value types supported by VBA can choose to truncate these values when not passed directly to such an external procedure.

If the referenced procedure was in a class module, the runtime semantics of expressions within that procedure that depend on the current instance, such as instance expressions, are implementation-defined.

6 VBA Standard Library

6.1 VBA Project

"VBA" is the *project name (section 4.1)* of a *host project (section 4.1)* that is present in every *VBA Environment*. The VBA project consists of a set of classes, functions, Enums and constants that form VBA's standard library.

6.1.1 Predefined Enums

6.1.1.1 FormShowConstants

Constant	Value
vbModal	1
vbModeless	0

6.1.1.2 VbAppWinStyle

Constant	Value
vbHide	0
vbMaximizedFocus	3
vbMinimizedFocus	2
vbMinimizedNoFocus	6
vbNormalFocus	1
vbNormalNoFocus	4

6.1.1.3 VbCalendar

Constant	Value
vbCalGreg	0
vbCalHijri	1

6.1.1.4 VbCallType

Constant	Value
vbGet	2
vbLet	4
vbMethod	1

Constant	Value
vbSet	8

6.1.1.5 VbCompareMethod

Constant	Value
vbBinaryCompare	0
vbTextCompare	1

6.1.1.6 VbDateTimeFormat

Constant	Value
vbGeneralDate	0
vbLongDate	1
vbLongTime	3
vbShortDate	2
vbShortTime	4

6.1.1.7 VbDayOfWeek

Constant	Value
vbFriday	6
vbMonday	2
vbSaturday	7
vbSunday	1
vbThursday	5
vbTuesday	3
vbUseSystemDayOfWeek	0
vbWednesday	4

6.1.1.8 VbFileAttribute

This Enum is used to encode the return value of the function VBA.Interaction.GetAttr.

Constant	Value	Description
vbNormal	0	Specifies files with no attributes.

Constant	Value	Description
vbReadOnly	1	Specifies read-only files.
vbHidden	2	Specifies hidden files.
VbSystem	4	Specifies system files.
vbVolume	8	Specifies volume label; if any other attributed is specified, vbVolume is ignored
vbDirectory	16	Specifies directories or folders.
vbArchive	32	Specifies files that have changed since the last backup.
vbAlias	64	Specifies file aliases on platforms that support them.

6.1.1.9 VbFirstWeekOfYear

Constant	Value
vbFirstFourDays	2
vbFirstFullWeek	3
vbFirstJan1	1
vbUseSystem	0

6.1.1.10 VbIMEStatus

Constant	Value
vbIMEAlphaDbl	7
vbIMEAlphaSng	8
vbIMEDisable	3
vbIMEHiragana	4
vbIMEKatakanaDbl	5
vbIMEKatakanaSng	6
vbIMEModeAlpha	8
vbIMEModeAlphaFull	7
vbIMEModeDisable	3
vbIMEModeHangul	10
vbIMEModeHangulFull	9
vbIMEModeHiragana	4
vbIMEModeKatakana	5
vbIMEModeKatakanaHalf	6

Constant	Value
vbIMEModeNoControl	0
vbIMEModeOff	2
vbIMEModeOn	1
vbIMENoOp	0
vbIMEOff	2
vbIMEOn	1

6.1.1.11 VbMsgBoxResult

Constant	Value
vbAbort	3
vbCancel	2
vbIgnore	5
vbNo	7
vbOK	1
vbRetry	4
vbYes	6

6.1.1.12 VbMsgBoxStyle

Constant	Value
vbAbortRetryIgnore	2
vbApplicationModal	0
vbCritical	16
vbDefaultButton1	0
vbDefaultButton2	256
vbDefaultButton3	512
vbDefaultButton4	768
vbExclamation	48
vbInformation	64
vbMsgBoxHelpButton	16384
vbMsgBoxRight	524288
vbMsgBoxRtlReading	1048576

Constant	Value
vbMsgBoxSetForeground	65536
vbOKCancel	1
vbOKOnly	0
vbQuestion	32
vbRetryCancel	5
vbSystemModal	4096
vbYesNo	4
vbYesNoCancel	3

6.1.1.13 VbQueryClose

Constant	Value
vbAppTaskManager	3
vbAppWindows	2
vbFormCode	1
vbFormControlMenu	0
vbFormMDIForm	4

6.1.1.14 VbStrConv

Constant	Value
vbFromUnicode	128
vbHiragana	32
vbKatakana	16
vbLowerCase	2
vbNarrow	8
vbProperCase	3
vbUnicode	64
vbUpperCase	1
vbWide	4

6.1.1.15 VbTriState

Constant	Value
vbFalse	0

Constant	Value
vbTrue	-1
vbUseDefault	-2

6.1.1.16 VbVarType

Constant	Value
vbArray	8192
vbBoolean	11
vbByte	17
vbCurrency	6
vbDataObject	13
vbDate	7
vbDecimal	14
vbDouble	5
vbEmpty	0
vbError	10
vbInteger	2
vbLong	3
vbLongLong	20 *(defined only on implementations that support a LongLong value type)*
vbNull	1
vbObject	9
vbSingle	4
vbString	8
vbUserDefinedType	36
vbVariant	12

6.1.2 Predefined Procedural Modules

Unless otherwise specified, all Predefined Procedural Modules in the VBA Standard Library defined with the attribute VB_GlobalNamespace set to "True" are *global modules*, allowing simple name access to their public constants, variables, and procedures as specified in section 5.6.10.

The following modules define their public constants as if they were defined using a <public-const-declaration>.

6.1.2.1 ColorConstants Module

Constant	Value
vbBlack	0
vbBlue	16711680
vbCyan	16776960
vbGreen	65280
vbMagenta	16711935
vbRed	255
vbWhite	16777215
vbYellow	65535

6.1.2.2 Constants Module

Constant	Value
vbBack	VBA.Strings.Chr$(8)
vbCr	VBA.Strings.Chr$(13)
vbCrLf	VBA.Strings.Chr$(13) + VBA.Strings.Chr$(10)
vbFormFeed	VBA.Strings.Chr$(12)
vbLf	VBA.Strings.Chr$(10)
vbNewLine	*An implementation-defined **String** value representing a new line*
vbNullChar	VBA.Strings.Chr$(0)
vbTab	VBA.Strings.Chr$(9)
vbVerticalTab	VBA.Strings.Chr$(11)
vbNullString	*An implementation-defined **String** value representing a null string pointer*
vbObjectError	-2147221504

6.1.2.3 Conversion Module

6.1.2.3.1 Public Functions

Note that these explicit-coercion functions are the only way to convert values from the **LongLong** type to any other type, as implicit conversions from **LongLong** to a declared type other than **LongLong** or **Variant** are not allowed.

6.1.2.3.1.1 CBool

Function Declaration

```
Function CBool(Expression As Variant) As Boolean
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **Integer** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **Boolean** *(section 5.5.1.2.2)*.
- If the value of Expression is not an **Error** *data value* return the **Boolean** *data value* that is the result of Expression being ***Let****-coerced* to **Boolean**.

6.1.2.3.1.2 CByte

Function Declaration

```
Function CByte(Expression As Variant) As Byte
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **Byte** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **Byte** *(section 5.5.1.2.1)*.
- If the value of Expression is not an **Error** *data value* return the **Byte** data value that is the result of Expression being ***Let***-coerced to **Byte**.

6.1.2.3.1.3 CCur

Function Declaration

```
Function CCur(Expression As Variant) As Currency
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **Currency** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **Currency** *(section 5.5.1.2.1)*.
- If the value of Expression is not an **Error** *data value* return the **Currency** data value that is the result of Expression being ***Let***-coerced to **Currency**.

6.1.2.3.1.4 CDate / CVDate

Function Declaration

```
Function CDate(Expression As Variant) As Date
Function CVDate(Expression As Variant)As Variant
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then raise error 13, "Type mismatch".
- If the value of Expression is not an **Error** *data value* return the **Date** data value that is the result of Expression being ***Let***-coerced to **Date** *(section 5.5.1.2.3)*.
- CDate MAY recognizes string date formats according to implementation defined locale settings.
- CVDate is identical to CDate except for the declared type of its return value.

6.1.2.3.1.5 CDbl

Function Declaration

```
Function CDbl(Expression As Variant) As Double
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **Double** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **Double** *(section 5.5.1.2.1)*.
- If the value of Expression is not an **Error** *data value* return the **Double** data value that is the result of Expression being ***Let***-coerced to **Double**.

6.1.2.3.1.6 CDec

Function Declaration

```
Function CDec(Expression As Variant)As Variant
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- Return the **Decimal** *data value* that is the result of Expression being ***Let****-coerced* to **Decimal** *(section 5.5.1.2.1)*.

6.1.2.3.1.7 CInt

Function Declaration

```
Function CInt(Expression As Variant) As Integer
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **Integer** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **Integer** *(section 5.5.1.2.1)*.
- If the value of Expression is not an **Error** *data value* return the **Integer** data value that is the result of Expression being ***Let***-coerced to **Integer**.

6.1.2.3.1.8 CLng

Function Declaration

```
Function CLng(Expression As Variant) As Long
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the *data value* of the **Long** *error code (section 2.1)* of the **Error** *data value*.
- If the value of Expression is not an **Error** *data value* return the **Long** data value that is the result of Expression being ***Let***-coerced to **Long** *(section 5.5.1.2.1)*.

6.1.2.3.1.9 CLngLng

Function Declaration

```
Function CLngLng(Expression As Variant) As LongLong
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **LongLong** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **LongLong**.
- If the value of Expression is not an **Error** *data value*, then return the **LongLong** data value that is the result of Expression being ***Let***-coerced to **LongLong**.

6.1.2.3.1.10 CLngPtr

Function Declaration

```
Function CLngPtr(Expression As Variant) As LongPtr
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **LongPtr** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **LongPtr**.

- If the value of Expression is not an **Error** *data value*, then return the **LongPtr** data value that is the result of Expression being ***Let***-coerced to **LongPtr**.

6.1.2.3.1.11 CSng

Function Declaration

```
Function CSng(Expression As Variant) As Single
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then return the **Single** data value that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* being ***Let****-coerced* to **Single** *(section 5.5.1.2.1)*.
- If the value of Expression is not **an Error** *data value* return th**e Single** d*ata value* that is the result of Expression being ***Let***-coerced to **Single**.

6.1.2.3.1.12 CStr

Functio*n* Declaration

```
Function CStr(Expression As Variant) As String
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then the returned value is the **String** *data value* consisting of "Error" followed by a single space character followed by the **String** that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* ***Let****-coerced* to **String** *(section 5.5.1.2.4)*.
- If the value of Expression is not an **Error** *data value* return the **String** data value that is the result of Expression being ***Let***-coerced to **String** *(section 5.5.1.2.4)*.

6.1.2.3.1.13 CVar

Function Declaration

```
Function CVar(Expression As Variant) As Variant
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- The argument *data value* is returned.

6.1.2.3.1.14 CVErr

Function Declaration

```
Function CVErr(Expression As Variant) As Variant
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the value of Expression is an **Error** *(section 2.1) data value* then the value of Expression is returned without raising an error.
- The *data value* of Expression is ***Let**-coerced* to **Long** *(section 5.5.1.2.1)* for use as an *error code (section 2.1)*. If the resulting data value is not in the inclusive range 0 to 65535, Error 5 is raised.
- Return an **Error** *(section 2.1) data value* whose *error code* is the result of Expression being ***Let***-coerced to **Long** *(section 5.5.1.2.1)*.

6.1.2.3.1.15 Error / Error$

Function Declaration

```
Function Error(Optional ErrorNumber)
Function Error$(Optional ErrorNumber) As String
```

Parameter	Description
Expression	Any *data value (section 2.1)*.

Runtime Semantics.

- If the parameter ErrorNumber is present its *data value* is ***Let***-coerced to **Long** *(section 5.5.1.2.1)* for use as an *error code (section 2.1)*. If the resulting *data value* is greater than 65,535 then Error 6 is raised. Negative values for ErrorNumber are acceptable.
- If the parameter ErrorNumber is not present, the most recently raised error number (or 0 if no error has been raised) is used as the *error code*. Note that the most recently raised error number might not necessarily be the same as the current value of Err.Number (section 6.1.3.2.2.5)
- The string data value returned by the function is determined based upon the error code as follows:
  - If the *error code* is 0 the data value is the zero length **String**.
  - If a descriptive text is specified for the *error code*, the *data value* is a **String** containing that descriptive text.
  - If the *error code* has an implementation specific meaning the descriptive text is also implementation specific.
  - Otherwise, the *data value* is "Application-defined or object-defined error."
- Error$ is identical to Error except for the declared type of its return value.

6.1.2.3.1.16 Fix

Returns the integer portion of a number.

Function Declaration

```
Function Fix(Number As Variant)
```

Parameter	Description
Number	Any *data value (section 2.1)*.

Runtime Semantics.

- If the *data value* of Number is **Null**, **Null** is returned.
- If the *value type (section 2.1)* of Number is **Integer**, **Long** *or* **LongLong**, the *data value* of Number is returned.
- If the *value type* of Number is any *numeric value type (section 5.5.1.2.1)* other than **Integer** or **Long**, the returned value is a *data value* whose *value type* is the same as the *value type* of Number and whose value that is the smallest integer greater than or equal to the *data value* of Number. If the value to be returned is not in the range of the *value type* of Number, raise error 6, "Overflow".
- If the *value type* of Number is **String**, the returned value is the result of the Fix function applied to the result of ***Let***-coercing Number to **Double.**
- If the *value type (section 2.1)* of Number is **Date**, the returned value is the same as result of evaluating the expression: CDate(Fix(CDbl(Number)))
- Otherwise, the returned value is the result of Number being ***Let***-coerced to **Integer**.

6.1.2.3.1.17 Hex / Hex$

Function Declaration

```
Function Hex(Number As Variant)
Function Hex$(Number As Variant) As String
```

Parameter	Description
Number	Any *data value (section 2.1)*.

Runtime Semantics.

- If the *data value* of the parameter Number is the data value **Null** the function Hex$ raises error 94, "Invalid use of Null" and the function Hex returns the *data value* **Null**.
- If the *data value* of the parameter Number is the data value **Empty** the function returns the **String** *data value* "0"
- If the *data value* of the parameter Number has the value type **LongLong**, it is not coerced.
- If the *data value* of the parameter Number is any other value, it is ***Let***-coerced to **Long** *(section 5.5.1.2.1)*.
- If the ***Let***-coerced value of Number is positive, the function result is a **String** *data value* consisting of the characters of the hexadecimal encoding with no leading zeros of the value.
- If the ***Let***-coerced value of Number is in the range -32,767 to -1, the function result is a four character **String** *data value* consisting of the characters of the 16-bit 2's complement hexadecimal encoding of the value.
- If the ***Let***-coerced value of Number is in the range -2,147,483,648 to -32,768, the function result is an eight character **String** *data value* consisting of the characters of the 32-bit 2's complement hexadecimal encoding of the value.
- If the ***Let***-coerced value of Number is in the range -9,223,372,036,854,775,808 to 2,147,483,649, the function result is a sixteen character **String** *data value* consisting of the characters of the 64-bit 2's complement hexadecimal encoding of the value.
- Except for the case where the parameter Number is **Null**, the semantics of Hex$ is identical to Hex except for the declared type of its returned value.

6.1.2.3.1.18 Int

Returns the integer portion of a number.

Function Declaration

```
Function Int(Number As Variant)
```

Parameter	Description
Number	Any *data value (section 2.1)*.

Runtime Semantics.

- If the *data value* of Number is **Null**, **Null** is returned.
- If the *value type (section 2.1)* of Number is **Integer**, **Long** or **LongLong**, the *data value* of Number is returned.
- If the *value type* of Number is any *numeric value type (section 5.5.1.2.1)* other than **Integer** or **Long**, the returned value is a *data value* whose *value type* is the same as the *value type* of Number and whose value that is the greatest integer that is less than or equal to the *data value* of Number. If the value to be returned is not in the range of the *value type* of Number, raise error 6, "Overflow".
- If the *value type* of Number is **String**, the returned value is the result of the Int function applied to the result of ***Let***-coercing Number to **Double.**
- If the *value type (section 2.1)* of Number is **Date**, the returned value is the same as result of evaluating the expression: CDate(Int(CDbl(Number)))
- Otherwise, the returned value is the result of Number being ***Let***-coerced to **Integer**.

6.1.2.3.1.19 Oct / Oct$

Function Declaration

```
Function Oct(Number As Variant)
Function Oct$(Number As Variant) As String
```

Parameter	Description
Number	Any *data value (section 2.1)*.

Runtime Semantics.

- If the *data value* of the parameter Number is the data value **Null** the function Oct$ raises error 94, "Invalid use of Null" and the function Oct returns the *data value* **Null**.
- If the *data value* of the parameter Number is the data value **Empty** the function returns the **String** *data value* "0".
- If the *data value* of the parameter Number has the value type **LongLong**, it is not coerced.
- If the *data value* of the parameter Number is any other value, it is ***Let***-coerced to **Long** *(section 5.5.1.2.1)*.
- If the ***Let***-coerced value of Number is positive, the function result is a **String** *data value* consisting of the characters of the hexadecimal encoding of the value with no leading zeros.

- If the ***Let***-coerced value of Number is in the range -32,767 to -1, the function result is a six character **String** *data value* consisting of the characters of the 16-bit 2's complement octal encoding of the value.
- If the ***Let***-coerced value of Number is in the range -2,147,483,648 to -32,768, the function result is an eleven character **String** *data value* consisting of the characters of the 32-bit 2's complement octal encoding of the value.
- If the ***Let***-coerced value of Number is in the range -9,223,372,036,854,775,808 to 2,147,483,649, the function result is a twenty-two character **String** *data value* consisting of the characters of the 64-bit 2's complement hexadecimal encoding of the value.
- Except for the case where the parameter Number is **Null**, the semantics of Oct$ is identical to Oct except for the declared type of its returned value.

6.1.2.3.1.20 Str / Str$

Function Declaration

```
Function Str(Number As Variant)
Function Str$(Number As Variant) As String
```

Parameter	Description
Number	Any *data value (section 2.1)*.

Runtime Semantics.

- If the *data value* of Number is **Null**, **Null** is returned.
- If the value of Number is an **Error** *(section 2.1) data value* then the returned value is the **String** *data value* consisting of "Error" followed by a single space character followed by the **String** that is the result of the **Long** *error code (section 2.1)* of the **Error** *data value* ***Let****-coerced* to **String** *(section 5.5.1.2.4)*.
- If the *value type* of Number is **Date**, the returned value is the result of ***Let****-coercing* Number to **String**.
- If the *data value* of Number is any *numeric value type*, let S be the result of ***Let****-coercing* Number to **String** using "." as the decimal separator. If the *data value* of Number is positive (or zero) the result is S with a single space character appended as its first character, otherwise the result is S.
- Otherwise, the returned value is the result of the Str function applied to the result of ***Let****-coercing* Number to **Double.**
- Str$ is identical to Str except for the declared type of its return value.

6.1.2.3.1.21 Val

Function Declaration

```
Function Val(Value As String) As Double
```

Parameter	Description
Value	Any **String** *data value (section 2.1)*.

Runtime Semantics.

- If Value is the 0 length **String** *data value* return the **Double** *data value* 0.
- Returns the numbers contained in a string as a **Double.**
- The Val function stops reading the string at the first character it can't recognize as part of a number. Symbols and characters that are often considered parts of numeric values, such as dollar signs and commas, are not recognized. However, the function recognizes the radix prefixes &O (for octal) and &H (for hexadecimal).
- Blanks, tabs, and linefeed characters are stripped from the argument.

6.1.2.4 DateTime Module

6.1.2.4.1 Public Functions

6.1.2.4.1.1 DateAdd

Function Declaration

```
Function DateAdd(Interval As String,
                 Number As Double,
                 Date As Variant)
```

Parameter	Description
Interval	**String** *data value (section 2.1)* that specifies the interval of time to add.
Number	The number of intervals to add. It can be positive (to get dates in the future) or negative (to get dates in the past). If it is not a integer value, it is rounded to the nearest whole number.
BaseDate	**Date,** *data value* to which the interval is added.

Runtime Semantics.

- The DateAdd function returns the result of adding or subtracting a specified time interval from a base date. For example, it can be used to calculate a date 30 days from today or a time 45 minutes from now.
- The interval argument is interpreted according to this table:

Interval Data Value	Meaning
"yyyy"	Year

Interval Data Value	Meaning
"q"	Quarter
"m"	Month
"y"	Day of year
"d"	Day
"w"	Weekday
"ww"	Week
"h"	Hour
"n"	Minute
"s"	Second
Any other data value	Raise Error 5, "Invalid procedure call or argument"

- The interpretation of the Interval *data value* is not case sensitive.
- The DateAdd function won't return an invalid date. The following example adds one month to January 31:

  DateAdd("m", 1, "31-Jan-95")

  In this case, DateAdd returns 28-Feb-95, not 31-Feb-95. If date is 31-Jan-96, it returns 29-Feb96 because 1996 is a leap year.
- If the calculated date would precede the year 100 (that is, you subtract more years than are in date), an error 5 is raised.
- For date, if the Calendar property setting is Gregorian, the supplied date MUST be Gregorian. If the calendar is Hijri, the supplied date MUST be Hijri. If month values are names, the name MUST be consistent with the current Calendar property setting. To minimize the possibility of month names conflicting with the current Calendar property setting, enter numeric month values (Short Date format).

6.1.2.4.1.2 DateDiff

Function Declaration

```
Function DateDiff(Interval As String,
                  Date1 As Variant,
                  Date2 As Variant,
                  Optional FirstDayOfWeek
                        As VbDayOfWeek = vbSunday,
                  Optional FirstWeekOfYear
                        As VbFirstWeekOfYear = vbFirstJan1
                  )
```

Parameter	Description
Interval	**String** *data value (section 2.1)* that specifies the interval of time to use to calculate the difference between Date1 and Date2.
Date1, Date2	The two dates to use in the calculation.
FirstDayOfWeek	A constant that specifies the first day of the week. If not specified, Sunday is assumed. See section 6.1.1.7.
FirstWeekOfYear	A constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.

Runtime Semantics.

- Returns a **Long** *data value* specifying the number of time intervals between two specified dates.
- The Interval argument is interpreted according to this table:

Interval Data Value	Meaning
"yyyy"	Year
"q"	Quarter
"m"	Month
"y"	Day of year
"d"	Day
"w"	Weekday
"ww"	Week
"h"	Hour
"n"	Minute
"s"	Second
Any other data value	Raise Error 5, "Invalid procedure call or argument"

- The interpretation of the Interval *data value* is not case sensitive.
- If Date1 falls on a Monday, DateDiff counts the number of Mondays until Date2. It counts Date2 but not Date1. If interval is Week ("ww"), however, the DateDiff function returns the number of calendar weeks between the two dates. It counts the number of Sundays between Date1 and Date2. DateDiff counts Date2 if it falls on a Sunday; but it doesn't count Date1, even if it does fall on a Sunday.
- If Date1 refers to a later point in time than Date2, the DateDiff function returns a negative number.
- The FirstDayOfWeek argument affects calculations that use the "w" and "ww" interval symbols.
- When comparing December 31 to January 1 of the immediately succeeding year, DateDiff for Year ("yyyy") returns 1 even though only a day has elapsed.

- For Date1 and Date2, if the Calendar property setting is Gregorian, the supplied date MUST be Gregorian. If the calendar is Hijri, the supplied date MUST be Hijri.

6.1.2.4.1.3 DatePart

Function Declaration

```
Function DatePart(Interval As String,
                  BaseDate As Variant,
                  Optional FirstDayOfWeek
                        As VbDayOfWeek = vbSunday,
                  Optional FirstWeekOfYear
                        As VbFirstWeekOfYear = vbFirstJan1
                  )
```

Parameter	Description
Interval	**String** *data value (section 2.1)* that specifies the interval of time to extract from BaseDate.
BaseDate	**Date** *data value* from which the interval is extracted.
FirstDayOfWeek	A constant that specifies the first day of the week. If not specified, Sunday is assumed. See section 6.1.1.7.
FirstWeekOfYear	A constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs.

Runtime Semantics.

- Returns a **Integer** *data value* containing the specified part of a given date
- The Interval argument is interpreted according to this table:

Interval Data Value	Meaning
"yyyy"	Year
"q"	Quarter
"m"	Month
"y"	Day of year
"d"	Day
"w"	Weekday
"ww"	Week
"h"	Hour
"n"	Minute
"s"	Second
Any other data value	Raise Error 5, "Invalid procedure call or argument"

- The interpretation of the Interval *data value* is not case sensitive.
- The FirstDayOfWeek argument affects calculations that use the "w" and "ww" interval symbols.
- For BaseDate, if the Calendar property setting is Gregorian, the supplied date MUST be Gregorian. If the calendar is Hijri, the supplied date MUST be Hijri.
- The returned date part is in the time period units of the current Arabic calendar. For example, if the current calendar is Hijri and the date part to be returned is the year, the year value is a Hijri year.

6.1.2.4.1.4 DateSerial

Function Declaration

```
Function DateSerial(Year As Integer, Month As Integer,
                    Day As Integer)
```

Parameter	Description
Year	An **Integer** *data value (section 2.1)* in the range 100 and 9999, inclusive.
Month	An **Integer** *data value (section 2.1)*.
Day	An **Integer** *data value (section 2.1)*.

Runtime Semantics.

- The DateSerial function returns a **Date** for a specified year, month, and day.
- To specify a date, such as December 31, 1991, the range of numbers for each DateSerial argument SHOULD be in the accepted range for the unit; that is, 1-31 for days and 1-12 for months. However, you can also specify relative dates for each argument using any numeric expression that represents some number of days, months, or years before or after a certain date.
- Two digit years for the year argument are interpreted based on implementation defined settings. The default settings are that values between 0 and 29, inclusive, are interpreted as the years 2000-2029. The default values between 30 and 99 are interpreted as the years 19301999. For all other year arguments, use a four-digit year (for example, 1800).
- When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 35 days, it is evaluated as one month and some number of days, depending on where in the year it is applied. If any single argument is outside the range -32,768 to 32,767, an error occurs. If the date specified by the three arguments falls outside the acceptable range of dates, an error occurs.
- For Year, Month, and Day, if the Calendar property setting is Gregorian, the supplied value is assumed to be Gregorian. If the Calendar property setting is Hijri, the supplied value is assumed to be Hijri.
- The returned date part is in the time period units of the current Visual Basic calendar. For example, if the current calendar is Hijri and the date part to be returned is the year, the year

value is a Hijri year. For the argument year, values between 0 and 99, inclusive, are interpreted as the years 1400-1499. For all other year values, use the complete four-digit year (for example, 1520).

6.1.2.4.1.5 DateValue

Function Declaration

```
Function DateValue(Date As String) As Variant
```

Parameter	Description
Date	**String** *data value (section 2.1)* representing a date from January 1, 100 through December 31, 9999. The value can also be a date, a time, or both a date and time.

Runtime Semantics.

- Returns a **Date** data value.
- If Date is a string that includes only numbers separated by valid date separators, DateValue recognizes the order for month, day, and year according to the implementation-defined Short Date format. DateValue also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, DateValue also recognizes December 30, 1991 and Dec 30, 1991.
- If the year part of Date is omitted, DateValue uses the current year from the system's date.
- If the Date argument includes time information, DateValue doesn't return it. However, if Date includes invalid time information (such as "89:98"), an error occurs.
- For Date, if the Calendar property setting is Gregorian, the supplied date MUST be Gregorian. If the calendar is Hijri, the supplied date MUST be Hijri. If the supplied date is Hijri, the argument date is a **String** representing a date from 1/1/100 (Gregorian Aug 2, 718) through 4/3/9666 (Gregorian Dec 31, 9999).

6.1.2.4.1.6 Day

Function Declaration

```
Function Day(Date As Variant) As Variant
```

Parameter	Description
Date	Any *data value (section 2.1)*. The *data value* SHOULD be **Let**-coercible to **Date**.

Runtime Semantics.

- Date is **Let**-coerced to **Date** and an **Integer** data value specifying a whole number between 1 and 31, inclusive, representing the day of the month is returned.

- If Date is Null, Null is returned.
- If the Calendar property setting is Gregorian, the returned **Integer** represents the Gregorian day of the month for the Date argument. If the calendar is Hijri, the returned **Integer** represents the Hijri day of the month for the Date argument.

6.1.2.4.1.7 Hour

Function Declaration

```
Function Hour(Time As Variant) As Variant
```

Parameter	Description
Time	Any *data value (section 2.1)*. The *data value* SHOULD be **Let**-coercible to **Date**.

Runtime Semantics.

- Time is **Let**-coerced to **Date** and an **Integer** specifying a whole number between 0 and 23, inclusive representing the hour of the day specified by the date is returned.
- If Time is **Null**, **Null** is returned.

6.1.2.4.1.8 Minute

Function Declaration

```
Function Hour(Time As Variant) As Variant
```

Parameter	Description
Time	Any *data value (section 2.1)*. The *data value* SHOULD be **Let**-coercible to **Date**.

Runtime Semantics.

- Time is **Let**-coerced to **Date** and an **Integer** specifying a whole number between 0 and 59, inclusive representing the minute of the hour specified by the date is returned.
- If Time is **Null**, **Null** is returned.

6.1.2.4.1.9 Month

Function Declaration

```
Function Month(Date As Variant) As Variant
```

Parameter	Description
Date	Any *data value (section 2.1)*. The *data value* SHOULD be **Let**-coercible to **Date**.

Runtime Semantics.

- Date is **Let**-coerced to **Date** and an **Integer** data value specifying a whole number between 1 and 12, inclusive, representing the month of the year is returned.
- If Date is Null, **Null** is returned.

6.1.2.4.1.10 Second

Function Declaration

```
Function Second(Time As Variant) As Variant
```

Parameter	Description
Time	Any *data value (section 2.1)*. The *data value* SHOULD be **Let**-coercible to **Date**.

Runtime Semantics.

- Time is **Let**-coerced to **Date** and an **Integer** specifying a whole number between 0 and 59, inclusive representing the second of the minute specified by the date is returned.
- If Time is **Null**, **Null** is returned.

6.1.2.4.1.11 TimeSerial

Function Declaration

```
Function TimeSerial(Hour As Integer,
                    Minute As Integer,
                    Second As Integer) As Variant
```

Parameter	Description
Hour	An **Integer** *data value (section 2.1)* in the range 0 and 23, inclusive.
Minute	An **Integer** *data value (section 2.1)*.
Second	An **Integer** *data value (section 2.1)*.

Runtime Semantics.

- Returns a **Date** containing the time for a specific hour, minute, and second.
- To specify a time, such as 11:59:59, the range of numbers for each TimeSerial argument SHOULD be in the normal range for the unit; that is, 023 for hours and 059 for minutes and seconds.

However, one can also specify relative times for each argument using any **Integer** data value that represents some number of hours, minutes, or seconds before or after a certain time.

- When any argument exceeds the normal range for that argument, it increments to the next larger unit as appropriate. For example, if Minute specifies 75 minutes, it is evaluated as one hour and 15 minutes. If the time specified by the three arguments causes the date to fall outside the acceptable range of dates, an error is raised.

6.1.2.4.1.12 TimeValue

Function Declaration

```
Function TimeValue(Time As String) As Variant
```

Parameter	Description
Time	**String** *data value (section 2.1)* representing a time from 0:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.), inclusive. The value can also be a date, a time, or both a date and time.

Runtime Semantics.

- Returns a **Date** containing the time. The argument string is **Let**-coerced to value type **Date** and the date portions of the data value are set to zero.
- If Time is **Null**, **Null** is returned.
- If the Time argument contains date information, TimeValue doesn't return it. However, if Time includes invalid date information, an error occurs.

6.1.2.4.1.13 Weekday

Function Declaration

```
Function Weekday(Date,
                 Optional FirstDayOfWeek
                       As VbDayOfWeek = vbSunday                  ) As Variant
```

Parameter	Description
Date	Any *data value (section 2.1)*. The *data value* SHOULD be **Let**-coercible to **Date**. If Date contains Null, Null is returned.
FirstDayOfWeek	A constant that specifies the first day of the week. If not specified, Sunday is assumed. See section 6.1.1.7.

Runtime Semantics.

- Returns an **Integer** containing a whole number representing the day of the week.
- The Weekday function can return any of these values (see section 6.1.1.7):

Constant	Value	Description
vbSunday	1	Sunday
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

- If the Calendar property setting is Gregorian, the returned **Integer** represents the Gregorian day of the week for the Date argument. If the calendar is Hijri, the returned **Integer** represents the Hijri day of the week for the Date argument. For Hijri dates, the argument number is any numeric expression that can represent a date and/or time from 1/1/100 (Gregorian Aug 2, 718) through 4/3/9666 (Gregorian Dec 31, 9999).

6.1.2.4.1.14 Year

Function Declaration

```
Function Year(Date As Variant) As Variant
```

Parameter	Description
Date	Any *data value (section 2.1)*. The *data value* SHOULD be **Let**-coercible to **Date**.

Runtime Semantics.

- Date is **Let**-coerced to **Date** and an **Integer** data value specifying a whole number between 100 and 9999, inclusive, representing the year is returned.
- If Date is **Null**, **Null** is returned.

6.1.2.4.2 Public Properties

6.1.2.4.2.1 Calendar

Property Declaration

```
Property Calendar As VbCalendar
```

Runtime Semantics.

- Returns or sets a value specifying the type of calendar to use by subsequent calls to the functions defined in section 6.1.2.4.

6.1.2.4.2.2 Date/Date$

Property Declaration

```
Property Date As Variant
Property Date$ As String
```

Runtime Semantics.

- Returns a **String** or a **Date** containing the current system date.
- Date, and if the calendar is Gregorian, Date$ behavior is unchanged by the Calendar property setting. If the calendar is Hijri, Date$ returns a 10-character string of the form mm-dd-yyyy, where mm (01-12), dd (01-30) and yyyy (1400-1523) are the Hijri month, day and year. The equivalent Gregorian range is Jan 1, 1980 through Dec 31, 2099.

6.1.2.4.2.3 Now

Property Declaration

```
Property Now As Variant
```

Runtime Semantics.

- Returns a **Date** data value specifying the current date and time.

6.1.2.4.2.4 Time/Time$

Property Declaration [Get Property]

```
Property Time As Variant
Property Time$ As String
```

Runtime Semantics.

- Returns a **String** or **Date** containing the current system time.

Property Declaration [Set Property]

```
Property Time As Variant
```

Runtime Semantics.

- Sets the system time.
- The value assigned to the Time property MUST be **Let**-coercible to a Date data value. The time portion of the Date data value is used to set the system time.

- If Time is a string, Time attempts to convert it to a time using the time separators specified for the system. If it can't be converted to a valid time, an error occurs.

6.1.2.4.2.5 Timer

Function Declaration

```
Property Timer As Single
```

Runtime Semantics.

- Returns a **Single** data value representing the number of seconds elapsed since midnight.
- The sub-second resolution is implementation dependent.

6.1.2.5 FileSystem

6.1.2.5.1 Public Functions

6.1.2.5.1.1 CurDir/CurDir$

```
Function CurDir(Optional Drive As Variant) As Variant
Function CurDir$(Optional Drive As Variant) As String
```

Parameter	Description
Drive	Optional **String** data value that identifiers an storage drive in an implementation defined manner.

Runtime Semantics.

- The valid format of a Drive **String** is implementation defined.
- If Drive is unspecified, or if Drive is a zero-length string, CurDir returns the current file path for the implementation-defined current drive as a **String** data value. If Drive validly identifies a storage drive, the current file path for that drive is returned a **String** data value.
- If the value of Drive is not a valid drive identifier, Error 68 ("Device Unavailable") is raised.

6.1.2.5.1.2 Dir

Function Declaration

```
Function Dir(Optional PathName As Variant,
             Optional Attributes
                    As VbFileAttribute = vbNormal             )As String
```

Parameter	Description
PathName	Any *data value (section 2.1)* that specifies a file name. It can include directory or folder, and drive. The *data value* SHOULD be **Let**-coercible to **String**. A zero-length string ("") is returned if PathName is not found.
Attributes	Constant or numeric expression, whose sum specifies file attributes. If omitted, returns files that match PathName but have no attributes.

Runtime Semantics.

- Returns a **String** data value representing the name of a file, directory, or folder that matches a specified pattern or file attribute, or the volume label of a drive.
- The attributes argument can be the logical or any combination of the values of the vbFileAttribute enumeration.
- Dir supports the use of multiple character (*) and single character (?) wildcards to specify multiple files.

6.1.2.5.1.3 EOF

Function Declaration

```
Function EOF(FileNumber As Integer) As Boolean
```

Parameter	Description
FileNumber	Any data value that is **Let**-coercible to declared type **Integer** and that is a valid *file number (section 5.4.5)*.

Runtime Semantics.

- Returns a **Boolean** data value indicating whether or not the current *file-pointer-position (section 5.4.5)* is at the end of a file that has been opened for Random or sequential Input.
- The EOF function returns False until the file-pointer-position is at the end of the file. With files opened for Random or Binary access, EOF returns False until the last executed Get statement is unable to read an entire record.
- Files opened for Output, EOF always returns True.

6.1.2.5.1.4 FileAttr

Function Declaration

```
Function FileAttr(FileNumber As Integer,
                  Optional ReturnType As Integer = 1                  ) As Long
```

Parameter	Description
FileNumber	An **Integer** data value that is a valid *file number (section 5.4.5)*.
ReturnType	An **Integer** data value that indicating the type of information to return. Specify the data value 1 to return a value indicating the file mode. The meaning of other data values is implementation defined.

Runtime Semantics.

- Returns a **Long** representing the file mode *(section 5.4.5)* for files opened using the Open statement.
- When the ReturnType argument is 1, the following return values indicate the file access mode:

Mode	Value
Input	1
Output	2
Random	4
Append	8
Binary	32

6.1.2.5.1.5 FileDateTime

Function Declaration

```
Function FileDateTime(PathName As String) As Variant
```

Parameter	Description
PathName	**String** expression that specifies a file name; can include directory or folder, and drive. An error is raised if PathName is not found.

Runtime Semantics.

- Returns a **Date** data value that indicates the date and time when a file was created or last modified.

6.1.2.5.1.6 FileLen

Function Declaration

```
Function FileLen(PathName As String) As Long
```

Parameter	Description
PathName	**String** expression that specifies a file name; can include directory or folder, and drive. An error is raised if PathName is not found.

Runtime Semantics.

- Returns a **Long** specifying the length of a file in bytes.
- If the specified file is open when the FileLen function is called, the value returned represents the size of the file immediately before it was opened.

6.1.2.5.1.7 FreeFile

Function Declaration

```
Function FreeFile(Optional RangeNumber As Variant                    ) As Integer
```

Parameter	Description
RangeNumber	**Integer** data value that specifies the range from which the next free *file number (section 5.4.5)* is to be returned. Specify the data value 0 (default) to return a file number in the range 1-255, inclusive. Specify the data value 1 to return a file number in the range 256-511, inclusive.

Runtime Semantics.

- Returns an **Integer** representing the next *file number* available for use by the Open statement.

6.1.2.5.1.8 Loc

Function Declaration

```
Function Loc(FileNumber As Integer) As Long
```

Parameter	Description
FileNumber	An **Integer** data value that is a valid *file number (section 5.4.5)*.

Runtime Semantics.

- Returns a **Long** specifying the current read/write position (in other words, the current *file-pointer-position (section 5.4.5)*) within an open file. The interpretation of the returned value depends upon the file access mode of the open file.

- The following describes the return value for each file access mode:

Mode	Return Value
Random	Number of the last record read from or written to the file.
Sequential	Current byte position in the file divided by 128. However, information returned by Loc for sequential files is neither used nor required.
Binary	Position of the last byte read or written.

6.1.2.5.1.9 LOF

Function Declaration

```
Function LOF(FileNumber As Integer) As Long
```

Parameter	Description
FileNumber	An **Integer** data value that is a valid *file number (section 5.4.5)*.

Runtime Semantics.

- Returns a **Long** representing the size, in bytes, of a file opened using the Open statement.

6.1.2.5.1.10 Seek

Function Declaration

```
Function Seek(FileNumber As Integer) As Long
```

Parameter	Description
FileNumber	An **Integer** data value that is a valid *file number (section 5.4.5)*.

Runtime Semantics.

- Returns a **Long** specifying the current read/write position (in other words, the file-current *file-pointer-position (section 5.4.5)*) within a file opened using the Open statement. This value will be between 1 and 2,147,483,647 (equivalent to 2^31 - 1), inclusive.
- The following describes the return values for each file access mode:

Mode	Return Value
Random	Number of the next record read or written.

Mode	Return Value
Binary, Output, Append, Input	Byte position at which the next operation takes place. The first byte in a file is at position 1, the second byte is at position 2, and so on.

6.1.2.5.2 Public Subroutines

6.1.2.5.2.1 ChDir

Subroutine Declaration

```
Sub ChDir(Path As String)
```

Parameter	Description
Path	**String** data value that identifies which directory or folder becomes the new default directory or folder. The path can include the drive. If no drive is specified, ChDir changes the default directory or folder on the current drive.

Runtime Semantics.

- ChDir changes the system's current directory or folder, but not the default drive.

6.1.2.5.2.2 ChDrive

Subroutine Declaration

```
Sub ChDrive(Drive As String)
```

Parameter	Description
Drive	**String** data value that specifies an existing drive. If Drive is a zero-length string (""), the current drive doesn't change. If the drive argument is a multiple-character string, ChDrive uses only the first letter.

Runtime Semantics.

- ChDrive changes the current default drive.

6.1.2.5.2.3 FileCopy

Subroutine Declaration

```
Sub FileCopy(Source As String, Destination As String)
```

Parameter	Description
Source	**String** data value that specifies the name of the file to be copied. The string can include directory or folder, and drive.
Destination	**String** data value that specifies the target file name. The string can include directory or folder, and drive.

Runtime Semantics.

- Copies a file in an implementation-defined manner.
- If the Source file is currently open, an error occurs.

6.1.2.5.2.4 Kill

Subroutine Declaration

```
Sub Kill(PathName)
```

Parameter	Description
PathName	**String** data value that specifies one or more file names to be deleted; can include directory or folder, and drive.

Runtime Semantics.

- Kill deletes files from a disk.
- Kill supports the use of multiple-character (*) and single-character (?) wildcards to specify multiple files.

6.1.2.5.2.5 MkDir

Subroutine Declaration

```
Sub MkDir(Path As String)
```

Parameter	Description
Path	**String** data value that identifies the directory or folder to be created. The path can include the drive. If no drive is specified, MkDir creates the new directory or folder on the current drive.

Runtime Semantics.

- MkDir creates a new directory or folder.

6.1.2.5.2.6 RmDir

Subroutine Declaration

```
Sub RmDir(Path As String)
```

Parameter	Description
Path	**String** data value that identifies the directory or folder to be removed. The path can include the drive. If no drive is specified, RmDir removes the directory or folder on the current drive.

Runtime Semantics.

- RmDir removes an existing directory or folder.
- An error occurs when using RmDir on a directory or folder containing files.

6.1.2.5.2.7 SetAttr

Subroutine Declaration

```
Sub SetAttr(PathName As String,
            Attributes As VbFileAttribute)
```

Parameter	Description
PathName	**String** data value that specifies a file name can include directory or folder, and drive.
Attributes	Constant or numeric expression, whose sum specifies file attributes.

Runtime Semantics.

- Sets attribute information for a file.
- A run-time error occurs when trying to set the attributes of an open file.

6.1.2.6 Financial

6.1.2.6.1 Public Functions

6.1.2.6.1.1 DDB

Function Declaration

```
Function DDB(Cost As Double, Salvage As Double,
             Life As Double, Period As Double,
             Optional Factor As Variant) As Double
```

Parameter	Description
Cost	**Double** specifying initial cost of the asset.
Salvage	**Double** specifying value of the asset at the end of its useful life.
Life	**Double** specifying length of useful life of the asset.
Period	**Double** specifying period for which asset depreciation is calculated.
Factor	**Double data value** specifying rate at which the balance declines. If omitted, the data value 2 (double-declining method) is assumed.

Runtime Semantics.

- Returns a **Double** data value specifying the depreciation of an asset for a specific time period using the double-declining balance method (or some other specified method).
- The Life and Period arguments MUST be expressed in the same units. For example, if Life is given in months, Period MUST also be given in months. All arguments MUST be positive numbers.
- The DDB function uses the following formula to calculate depreciation for a given period:

Depreciation / Period = ((Cost - Salvage) * Factor) / Life

6.1.2.6.1.2 FV

Function Declaration

```
Function FV(Rate As Double, NPer As Double, Pmt As Double,            PV As Variant, Due As
Variant) As Double
```

Parameter	Description
Rate	**Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.
NPer	**Integer** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
Pmt	**Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.
Pv	**Double** data value specifying present value (or lump sum) of a series of future payments. For example, when borrowing money to buy a car, the loan amount is the present value to the lender of the monthly car payments that will be made. If omitted, the data value 0 is assumed.
Type	**Integer** data value specifying when payments are due. Use the data value 0 if payments are due at the end of the payment period, or use the data value 1 if payments are due at the beginning of the period. If omitted, the data value 0 is assumed.

Runtime Semantics.

- Returns a **Double** specifying the future value of an annuity based on periodic, fixed payments and a fixed interest rate.
- The Rate and NPer arguments MUST be calculated using payment periods expressed in the same units. For example, if Rate is calculated using months, NPer MUST also be calculated using months.
- For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

6.1.2.6.1.3 IPmt

Function Declaration

```
Function IPmt(Rate As Double, Per As Double,
              NPer As Double, PV As Double,
              Optional FV As Variant,
              Optional Due As Variant) As Double
```

Parameter	Description
Rate	**Double** data value specifying interest rate per period. For example, given a car loan at an annual percentage rate (APR) of 10 percent and making monthly payments, the rate per period is 0.1/12, or 0.0083.
Per	**Double** data value specifying payment period in the range 1 through NPer.
NPer	**Double** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.
Pv	**Double** data value specifying present value, or value today, of a series of future

Parameter	Description
	payments or receipts.
Fv	**Double** data value specifying future value or cash balance desired after final payment has been made. For example, the future value of a loan is $0 because that's its value after the final payment. However, if someone wants to save $50,000 over 18 years for their child's education, then $50,000 is the future value. If omitted, the data value 0.0 is assumed.
Type	**Integer** data value specifying when payments are due. Use the data value 0 if payments are due at the end of the payment period, or use the data value 1 if payments are due at the beginning of the period. If omitted, the data value 0 is assumed.

Runtime Semantics.

- Returns a **Double** specifying the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.
- The Rate and NPer arguments MUST be calculated using payment periods expressed in the same units. For example, if Rate is calculated using months, NPer MUST also be calculated using months.
- For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

6.1.2.6.1.4 IRR

Function Declaration

```
Function IRR(ValueArray() As Double,
             Optional Guess As Variant) As Double
```

Parameter	Description
Values	Array of **Double** data values specifying cash flow values. The array MUST contain at least one negative value (a payment) and one positive value (a receipt).
Guess	**Double** data value specifying estimated value that will be returned by IRR. If omitted, Guess is the data value 0.1 (10 percent).

Runtime Semantics.

- Returns a **Double** data value specifying the internal rate of return for a series of periodic cash flows (payments and receipts).
- The internal rate of return is the interest rate received for an investment consisting of payments and receipts that occur at regular intervals.

- The IRR function uses the order of values within the array to interpret the order of payments and receipts. The cash flow for each period doesn't have to be fixed, as it is for an annuity.
- IRR is calculated by iteration. Starting with the value of guess, IRR cycles through the calculation until the result is accurate to within 0.00001 percent. If IRR can't find a result after 20 tries, it fails.

6.1.2.6.1.5 MIRR

Function Declaration

```
Function MIRR(ValueArray() As Double,
              Finance_Rate As Double,
              Reinvest_Rate As Double) As Double
```

Parameter	Description
Values	Array of **Double** data values specifying cash flow values. The array MUST contain at least one negative value (a payment) and one positive value (a receipt).
Finance_Rate	**Double** data value specifying interest rate paid as the cost of financing.
Reinvest_Rate	**Double** data value specifying interest rate received on gains from cash reinvestment.

Runtime Semantics.

- Returns a **Double** data value specifying the modified internal rate of return for a series of periodic cash flows (payments and receipts).
- The modified internal rate of return is the internal rate of return when payments and receipts are financed at different rates. The MIRR function takes into account both the cost of the investment (Finance_Rate) and the interest rate received on reinvestment of cash (Reinvest_Rate).
- The Finance_Rate and Reinvest_Rate arguments are percentages expressed as decimal values. For example, 12 percent is expressed as 0.12.
- The MIRR function uses the order of values within the array to interpret the order of payments and receipts.

6.1.2.6.1.6 NPer

Function Declaration

```
Function NPer(Rate As Double, Pmt As Double, PV As Double,
              Optional FV As Variant,
              Optional Due As Variant) As Double
```

Parameter	Description
Rate	**Double** data value specifying interest rate per period. For example, given a loan at an annual percentage rate (APR) of 10 percent and making monthly payments, the rate per period is 0.1/12, or 0.0083.
Pmt	**Double** data value specifying payment to be made each period.
Pv	**Double** specifying present value, or value today, of a series of future payments or receipts.
Fv	**Double** data value specifying future value or cash balance desired after final payment has been made. If omitted, the **Double** data value 0.0 is assumed.
Type	**Integer** data value specifying when payments are due. Use the data value 0 if payments are due at the end of the payment period, or use the data value 1 if payments are due at the beginning of the period. If omitted, the data value 0 is assumed.

Runtime Semantics.

- Returns a **Double** data value specifying the number of periods for an annuity based on periodic, fixed payments and a fixed interest rate.
- For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

6.1.2.6.1.7 NPV

Function Declaration

```
Function NPV(Rate As Double, ValueArray() As Double) As Double
```

Parameter	Description
Rate	**Double** data value specifying discount rate over the length of the period, expressed as a decimal fraction.
Values	Array of **Double** data values specifying cash flow values. The array MUST contain at least one negative value (a payment) and one positive value (a receipt).

Runtime Semantics.

- Returns a **Double** data value specifying the net present value of an investment based on a series of periodic cash flows (payments and receipts) and a discount rate.
- The NPV function uses the order of values within the array to interpret the order of payments and receipts.
- The NPV investment begins one period before the date of the first cash flow value and ends with the last cash flow value in the array.

- The net present value calculation is based on future cash flows. If the first cash flow occurs at the beginning of the first period, the first value MUST be added to the value returned by NPV and MUST NOT be included in the cash flow values of Values( ).
- The NPV function is similar to the PV function (present value) except that the PV function allows cash flows to begin either at the end or the beginning of a period. Unlike the variable NPV cash flow values, PV cash flows MUST be fixed throughout the investment.

6.1.2.6.1.8 Pmt

Function Declaration

```
Function Pmt(Rate As Double, NPer As Double, PV As Double,
             Optional FV As Variant,
             Optional Due As Variant) As Double
```

Parameter	Description
Rate	**Double** data value specifying interest rate per period as a decimal fraction.
NPer	**Integer** data value specifying total number of payment periods in the annuity.
Pv	**Double** data value specifying present value (or lump sum) that a series of payments to be paid in the future is worth now.
Fv	**Double** data value specifying future value or cash balance desired after the final payment has been made. If omitted, the data value 0.0 is assumed.
Type	**Integer** data value specifying when payments are due. Use the data value 0 if payments are due at the end of the payment period, or use the data value1 if payments are due at the beginning of the period. If omitted, the data value 0 is assumed.

Runtime Semantics.

- Returns a **Double** data value specifying the payment for an annuity based on periodic, fixed payments and a fixed interest rate.
- The Rate and NPer arguments MUST be calculated using payment periods expressed in the same units. For example, if Rate is calculated using months, NPer MUST also be calculated using months.
- For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

6.1.2.6.1.9 PPmt

Function Declaration

```
Function PPmt(Rate As Double, Per As Double,
              NPer As Double, PV As Double,
              Optional FV As Variant,
```

```
            Optional Due As Variant) As Double
```

Parameter	Description
Rate	**Double** data value specifying interest rate per period. For example, given a loan at an annual percentage rate (APR) of 10 percent and making monthly payments, the rate per period is 0.1/12, or 0.0083.
Per	**Integer** data value specifying payment period in the range 1 through NPer.
NPer	**Integer** data value specifying total number of payment periods in the annuity. For example, if making monthly payments on a four-year loan, the loan has a total of 4 * 12 (or 48) payment periods.
Pv	**Double** data value specifying present value, or value today, of a series of future payments or receipts.
Fv	**Double** data value specifying future value or cash balance desired after the final payment has been made. If omitted, the data value 0.0 is assumed.
Type	**Integer** data value specifying when payments are due. Use the data value 0 if payments are due at the end of the payment period, or use the data value1 if payments are due at the beginning of the period. If omitted, the data value 0 is assumed.

Runtime Semantics.

- Returns a Double data value specifying the principal payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate. The Rate and NPer arguments MUST be calculated using payment periods expressed in the same units. For example, if Rate is calculated using months, NPer MUST also be calculated using months.
- For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

6.1.2.6.1.10 PV

Function Declaration

```
Function PV(Rate As Double, NPer As Double, Pmt As Double,
            Optional FV As Variant,
            Optional Due As Variant) As Double
```

Parameter	Description
Rate	**Double** data value specifying interest rate per period. For example, given a loan at an annual percentage rate (APR) of 10 percent and making monthly payments, the rate per period is 0.1/12, or 0.0083.

Parameter	Description
NPer	**Integer** data value specifying total number of payment periods in the annuity.
Pmt	**Double** data value specifying present value (or lump sum) that a series of payments to be paid in the future is worth now.
Fv	**Double** data value specifying future value or cash balance desired after the final payment has been made.
Type	**Integer** data value specifying when payments are due. Use the data value 0 if payments are due at the end of the payment period, or use the data value 1 if payments are due at the beginning of the period. If omitted, the data value 0 is assumed.

Runtime Semantics.

- Returns a Double data value specifying the present value of an annuity based on periodic, fixed payments to be paid in the future and a fixed interest rate.
- The Rate and NPer arguments MUST be calculated using payment periods expressed in the same units. For example, if Rate is calculated using months, NPer MUST also be calculated using months.
- For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

6.1.2.6.1.11 Rate

Function Declaration

```
Function Rate(NPer As Double, Pmt As Double, PV As Double,
              Optional FV As Variant,
              Optional Due As Variant,
              Optional Guess As Variant) As Double
```

Parameter	Description
NPer	**Double** data value specifying total number of payment periods in the annuity.
Pmt	**Double** data value specifying payment to be made each period.
Pv	**Double** data value specifying present value, or value today, of a series of future payments or receipts.
Fv	**Double** data value specifying future value or cash balance desired after the final payment has been made. If omitted, the data value 0.0 is assumed.
Type	**Integer** data value specifying when payments are due. Use the data value 0 if payments are due at the end of the payment period, or use the data value1 if payments are due at the beginning of the period. If omitted, the data value 0 is assumed.

Parameter	Description
Guess	**Double** data value specifying the estimated value that will be returned by Rate. If omitted, guess is the data value 0.1 (10 percent).

Runtime Semantics.

- Returns a **Double** data value specifying the interest rate per period for an annuity.
- For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.
- Rate is calculated by iteration. Starting with the value of Guess, Rate cycles through the calculation until the result is accurate to within 0.00001 percent. If Rate can't find a result after 20 tries, it fails.

6.1.2.6.1.12 SLN

Function Declaration

```
Function SLN(Cost As Double, Salvage As Double,
             Life As Double) As Double
```

Parameter	Description
Cost	**Double** data value specifying initial cost of the asset.
Salvage	**Double** data value specifying value of the asset at the end of its useful life.
Life	**Double** data value specifying length of useful life of the asset.

Runtime Semantics.

- Returns a **Double** data value specifying the straight-line depreciation of an asset for a single period.
- The depreciation period MUST be expressed in the same unit as the life argument. All arguments MUST be positive numbers.

6.1.2.6.1.13 SYD

Function Declaration

```
Function SYD(Cost As Double, Salvage As Double,
             Life As Double, Period As Double) As Double
```

Parameter	Description
Cost	**Double** data value specifying initial cost of the asset.
Salvage	**Double** data value specifying value of the asset at the end of its useful life.
Life	**Double** data value specifying length of useful life of the asset.
Period	**Double** data value specifying period for which asset depreciation is calculated.

Runtime Semantics.

- Returns a **Double** data value specifying the sum-of-years' digits depreciation of an asset for a specified period.
- The Life and Period arguments MUST be expressed in the same units. For example, if Life is given in months, period MUST also be given in months. All arguments MUST be positive numbers.

6.1.2.7 Information

6.1.2.7.1 Public Functions

6.1.2.7.1.1 IMEStatus

Function Declaration

```
Function IMEStatus() As VbIMEStatus
```

Runtime Semantics.

- Returns an **Integer** data value specifying the current implementation dependent Input Method Editor (IME) mode.

6.1.2.7.1.2 IsArray

Function Declaration

```
Function IsArray(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Data value to test to see if it is an array.

Runtime Semantics.

- IsArray returns **True** if the data value of Arg is an array data value; otherwise, it returns **False**

6.1.2.7.1.3 IsDate

Function Declaration

```
Function IsDate(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Data value to test to see if it is a Date.

Runtime Semantics.

- Returns a Boolean value indicating whether Arg is a **Date** data value or a **String** data value that can be **Let**-coerced to a **Date** data value.

6.1.2.7.1.4 IsEmpty

Function Declaration

```
Function IsEmpty(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Any data value.

Runtime Semantics.

- IsEmpty returns **True** if the data value of Arg is the data value **Empty**. Otherwise, it returns **False**.

6.1.2.7.1.5 IsError

Function Declaration

```
Function IsError(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Any data value.

Runtime Semantics.

- IsEmpty returns **True** if the data value of Arg is an **Error** data value. Otherwise, it returns **False**.

6.1.2.7.1.6 IsMissing

Function Declaration

```
Function IsMissing(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Any data value.

Runtime Semantics.

- IsMissing returns **True** if the data value of Arg is the **Missing** data value. Otherwise, it returns **False**.
- If IsMissing is used on a ParamArray argument, it always returns **False**.

6.1.2.7.1.7 IsNull

Function Declaration

```
Function IsNull(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Any data value.

Runtime Semantics.

- IsNull returns **True** if the data value of Arg is the **Null** data value. Otherwise, it returns **False**.

6.1.2.7.1.8 IsNumeric

Function Declaration

```
Function IsNumeric(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Any data value.

Runtime Semantics.

- IsNumeric returns **True** if the value type of the data value of Arg is any of **Byte**, **Currency**, **Decimal**, **Double**, **Integer**, **Long**, **LongLong** or **Single**. Otherwise, it returns **False**.

6.1.2.7.1.9 IsObject

Function Declaration

```
Function IsObject(Arg As Variant) As Boolean
```

Parameter	Description
Arg	Any data value.

Runtime Semantics.

- Returns **True** if the value type of the data value of Arg is **Object Reference**. Otherwise, it returns **False**.

6.1.2.7.1.10 QBColor

Function Declaration

```
Function QBColor(Color As Integer) As Long
```

Parameter	Description
Color	**Integer** data value in the range 0-15.

Runtime Semantics.

- If the data value of Color is outside of the range 0-15 then Error 5 ("Invalid procedure call or argument") is raised.
- The color argument represents color values used by earlier versions of Visual Basic. Starting with the least-significant byte, the returned value specifies the red, green, and blue values used to set the appropriate color in the RGB system used by Visual Basic for Applications.
- If the return value is specified by the following table:

Color data value	Returned data value	Common name of color
0	0	Black
1	&H800000	Blue
2	&H8000	Green
3	&H808000	Cyan

Color data value	Returned data value	Common name of color
4	&H80	Red
5	&H800080	Magenta
6	&H8080	Yellow
7	&HC0C0C0	White
8	&H808080	Gray
9	&HFF0000	Light Blue
10	&HFF00	Light Green
11	&HFFFF00	Light Cyan
12	&HFF	Light Red
13	&HFF00FF	Light Magenta
14	&HFFFF	Light Yellow
15	&HFFFFFF	Bright White

6.1.2.7.1.11 RGB

Function Declaration

```
Function RGB(Red As Integer, Green As Integer,
             Blue As Integer) As Long
```

Parameter	Description
Red	**Integer** data value in the range 0-255, inclusive, that represents the red component of the color.
Green	**Integer** data value in the range 0-255, inclusive, that represents the green component of the color.
Blue	**Integer** data value in the range 0-255, inclusive, that represents the blue component of the color.

Runtime Semantics.

- Returns the Long data value:

  (max(Blue,255)*65536)+(max(Green,255)*256)+max(Red,255).

6.1.2.7.1.12 TypeName

Function Declaration

```
Function TypeName(Arg As Variant) As String
```

Parameter	Description
Arg	Any data value.

Runtime Semantics.

- Returns a **String** that provides information about a variable.
- The string returned by TypeName can be any one of the following:

Value type of data value of Arg	String data value returned
An object whose type is **Object**	The name of the object type
Byte	"Byte"
Integer	"Integer"
Long	"Long"
LongLong	"LongLong"
Single	"Single"
Double	"Double"
Currency	"Currency"
Decimal	"Decimal"
Date	"Date"
String	"String"
Boolean	"Boolean"
An error value or **Missing**	"Error"
Empty	"Empty"
Null	"Null"
Any **Object Reference** except **Nothing**	"Object"
An object whose type is unknown	"Unknown"
Nothing	"Nothing"

- If Arg is an array, the returned string can be any one of the possible returned strings (or Variant) with empty parentheses appended. For example, if Arg is an array of **Integer**, TypeName returns "Integer()". If Arg is an array of **Variant** values, TypeName returns "Variant()".

6.1.2.7.1.13 VarType

Function Declaration

```
Function VarType(VarName As Variant) As VbVarType
```

Parameter	Description
VarName	Any data value.

Runtime Semantics.

- Returns an **Integer** indicating the subtype of a variable.
- The required VarName argument is a Variant containing any variable except a variable of a user-defined type.
- Returns a value from the following table based on VarName's value type:

VarName's value type	Value
Any Array type	8192 + *VarType of element's type*
Boolean	11
Byte	17
Currency	6
Date	7
Decimal	14
Double	5
Empty	0
Error *or* **Missing**	10
Integer	2
Long	3
LongLong *(defined only on implementations that support a LongLong value type)*	20
Null	1
Object reference	9
Single	4
String	8
Any UDT	36 *when the declared type is Variant.* 0 *when the declared type is a UDT.*
Variant *(as an element type of an array)*	12
An implementation-defined value that can be stored in a **Variant** *but that has no value in VBA*	13

6.1.2.8 Interaction

6.1.2.8.1 Public Functions

6.1.2.8.1.1 CallByName

Function Declaration

```
Function CallByName(Object As Object, ProcName As String, CallType As VbCallType, Args() As
Variant)
```

Parameter	Description
Object	**Object** containing the object on which the function will be executed.
ProcName	**String** containing the name of a property or method of the object.
CallType	A constant of type vbCallType representing the type of procedure being called.
Args()	**Variant** array containing arguments to be passed to the method.

Runtime Semantics.

- The CallByName function is used to get or set a property, or invoke a method at run time using a string name, based on the value of the CallType argument:

Constant	Value	Action
vbGet	2	Property Get
vbLet	4	Property Let
vbMethod	1	Method invocation
vbSet	8	Property Set

- If CallType has the value vbSet, the last argument in the Args array represents the value to set.

6.1.2.8.1.2 Choose

Function Declaration

```
Function Choose(Index As Single, ParamArray Choice() As Variant)
```

Parameter	Description
Index	Numeric expression that results in a value between the data value 1 and the number of available choices.
Choice	**A ParamArray argument** containing all the functions arguments starting with the second argument.

Runtime Semantics.

- Returns a value from its list of arguments.
- Choose returns a value from the list of choices based on the value of index. If Index is n, Choose returns the n-th element of the Choice ParamArray.
- The Choose function returns the data value **Null** if Index is less than 1 or greater than the number of choices listed.
- If Index argument is **Let**-coerced to declared type Integer before being used to select

6.1.2.8.1.3 Command

Function Declaration

```
Function Command() As Variant
Function Command$() As String
```

Runtime Semantics.

- Returns the argument portion of the implementation dependent command used to initiate execution of the currently executing VBA program.
- The runtime semantics of Command$ are identical to those of Command with the exception that the declared type of the return value is **String** rather than **Variant**.

6.1.2.8.1.4 CreateObject

Function Declaration

```
Function CreateObject(Class As String, Optional ServerName
As String)
```

Parameter	Description
Class	A **String** data value, containing the application name and class of the object to create.
ServerName	A **String** data value, containing the name of the network server where the object will be created. If ServerName is an empty string (""), the local machine is used.

Runtime Semantics.

- Creates and returns an object reference to an externally provided and possibly remote object.
- The class argument uses the Function Declaration AppName.ObjectType and has these parts:

Parameter	Description
AppName	The name of the application providing the object. The form and interpretation of an AppName is implementation defined.
ObjectType	The name of the type or class of object to create. The form and interpretation of an ObjectType name is implementation defined.

- The data value returned by CreateObject is an object reference and can be used in any context where an object reference is expected.
- If remote objects are supported it is via an implementation defined mechanism.
- The format and interpretation of the ServerName argument is implementation defined but the intent is to identify a specific remote computer that that is responsible for providing a reference to a remote object.
- An implementation can provide implementation defined mechanisms for designating single instance classes in which case only one instance of such a class is created, no matter how many times CreateObject is call requesting an instance of such a class.

6.1.2.8.1.5 DoEvents

Function Declaration

```
Function DoEvents() As Integer
```

Runtime Semantics.

- Yields execution so that the operating system can process externally generated events.
- The DoEvents function returns an **Integer** with an implementation defined meaning.
- DoEvents passes control to the operating system. Control is returned after the operating system has finished processing any events in its queue and all keys in the SendKeys queue have been sent.

6.1.2.8.1.6 Environ / Environ$

Function Declaration

```
Function Environ(Key As Variant) As Variant
Function Environ$(Key As Variant) As Variant
```

Parameter	Description
Key	Either a **String** or a data value that is **Let**-coercible to **Long**

Runtime Semantics.

- Returns the **String** associated with an implementation-defined environment variable.

- If Key is a **String** and is not the name of a defined environment variable, a zero-length string ("") is returned. Otherwise, Environ returns the string value of the environment variable whose name is the value of Key.
- If Key is numeric the string occupying that numeric position in the environment-string table is returned. The first value in the table starts at position 1. In this case, Environ returns a string of the form "name=value" where name is the name of the environment variable and value is its value. If there is no environment string in the specified position, Environ returns a zero-length string.
- The runtime semantics of Environ$ are identical to those of Environ with the exception that the declared type of the return value is **String** rather than **Variant**.

6.1.2.8.1.7 GetAllSettings

Function Declaration

```
Function GetAllSettings(AppName As String, Section As String)
```

Parameter	Description
AppName	**String** expression containing the name of the application or project whose key settings are requested.
Section	**String** expression containing the name of the section whose key settings are requested.

Runtime Semantics.

- If either AppName or Section does not exist in the settings store, return the data value Empty.
- Returns a two-dimensional array of strings containing all the key settings in the specified section and their corresponding values. The lower bound of each dimension is 1. The upper bound of the first dimension is the number of key/value pair. The upper bound of the second dimension is 2.

6.1.2.8.1.8 GetAttr

Function Declaration

```
Function GetAttr(PathName As String) As VbFileAttribute
```

Parameter	Description
PathName	Expression that specifies a file name; can include directory or folder, and drive.

Runtime Semantics.

- The argument MUST be a valid implementation defined external file identifier.
- Returns an **Integer** representing attributes of the file, directory, or folder identified by PathName.

- The value returned by GetAttr is composed of the sum of the following of the Enum elements of the Enum VBA.VbFileAttribute and have the following meanings:

Constant	Value	Description
vbNormal	0	Normal.
vbReadOnly	1	Read-only.
vbHidden	2	Hidden.
vbSystem	4	System file.
vbDirectory	16	Directory or folder.
vbArchive	32	File has changed since last backup.

6.1.2.8.1.9 GetObject

Function Declaration

```
Function GetObject(Optional PathName As Variant, Optional Class As Variant)
```

Parameter	Description
Class	**String**, containing the application name and class of the object to create.
PathName	**String**, containing the name of the network server where the object will be created. If PathName is an empty string (""), the local machine is used.

Runtime Semantics.

- Returns an object reference to an externally provided and possibly remote object.
- The Class argument uses the syntax AppName.ObjectType and has these parts:

Parameter	Description
AppName	The name of the application providing the object. The form and interpretation of an AppName is implementation defined.
ObjectType	The name of the type or class of object to create. The form and interpretation of an ObjectType name is implementation defined.

- Returns an object reference to an externally provided and possibly remote object.
- If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times CreateObject is executed. With a single-instance object, GetObject always returns the same instance when called with the zero-length string ("") syntax,

and it causes an error if the pathname argument is omitted. You can't use GetObject to obtain a reference to a class created with Visual Basic.

6.1.2.8.1.10 GetSetting

Function Declaration

```
Function GetSetting(AppName As String, Section As String, Key As String, Optional Default As
Variant) As String
```

Parameter	Description
AppName	**String** expression containing the name of the application or project whose key setting is requested.
Section	**String** expression containing the name of the section where the key setting is found.
Key	**String** expression containing the name of the key setting to return.
Default	**Variant** expression containing the value to return if no value is set in the key setting. If omitted, default is assumed to be a zero-length string ("").

Runtime Semantics.

- Returns a key setting value from an application's entry in an implementation dependent application registry.
- If any of the items named in the GetSetting arguments do not exist, GetSetting returns the value of Default.

6.1.2.8.1.11 IIf

Function Declaration

```
Function IIf(Expression As Variant, TruePart As Variant, FalsePart As Variant) As Variant
```

Parameter	Description
Expression	**Variant** containing the expression to be evaluated.
TruePart	**Variant**, containing the value to be returned if Expression evaluates to the data value True.
FalsePart	**Variant**, containing the value to be returned if Expression evaluates to the data value False.

Runtime Semantics.

- Returns one of two parts, depending on the evaluation of an expression.

- IIf always evaluates both TruePart (first) and FalsePart, even though it returns only one of them. For example, if evaluating FalsePart results in a division by zero error, an error occurs even if Expression is True.

6.1.2.8.1.12 InputBox

Function Declaration

```
Function InputBox(Prompt As Variant, Optional Title As
Variant, Optional Default As Variant, Optional XPos As
Variant, Optional YPos As Variant, Optional HelpFile As
Variant, Optional Context As Variant) As String
```

Parameter	Description
Prompt	**String** data value to be displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used. If prompt consists of more than one line, the lines can be separated using a carriage return character (Chr(13)), a linefeed character (Chr(10)), or carriage return + linefeed character combination (Chr(13) & Chr(10)) between each line.
Title	**String** to be displayed in the title bar of the dialog box. If Title is omitted, the *project name*(4.1) is placed in the title bar.
Default	**String** to be displayed in the text box as the default response if no other input is provided. If Default is omitted, the text box is displayed empty.
XPos	**Long** that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If XPos is omitted, the dialog box is horizontally centered.
YPos	**Long** that specifies, in twips, the vertical distance of the upper edge of the dialog box from the top of the screen. If YPos is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.
HelpFile	**String** that identifies the Help file to use to provide context-sensitive Help for the dialog box. If HelpFile is provided, Context MUST also be provided.
Context	**Long** that is the Help context number assigned to the appropriate Help topic by the Help author. If Context is provided, HelpFile MUST also be provided.

Runtime Semantics.

- Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns a **String** containing the contents of the text box.
- When both HelpFile and Context are provided, the user can press F1 to view the Help topic corresponding to the context. Some *host applications* can also automatically add a Help button to the dialog box. If the user clicks OK or presses ENTER , the InputBox function returns whatever is in the text box. If the user clicks Cancel, the function returns a zero-length string ("").

- Note: to specify more than the first named argument, you MUST use InputBox in an expression.

  To omit some positional arguments, you MUST include the corresponding comma delimiter.

6.1.2.8.1.13 MsgBox

Function Declaration

```
Function MsgBox(Prompt As Variant, Optional Buttons As
VbMsgBoxStyle = vbOKOnly, Optional Title As Variant,
Optional HelpFile As Variant, Optional Context As Variant) As VbMsgBoxResult
```

Parameter	Description
Prompt	**String** to be displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used. If prompt consists of more than one line, the lines can be separated using a carriage return character (Chr(13)), a linefeed character (Chr(10)), or carriage return + linefeed character combination (Chr(13) & Chr(10)) between each line.
Buttons	Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for Buttons is 0.
Title	**String** to be displayed in the title bar of the dialog box. If Title is omitted, the *project name (section 4.1)* is placed in the title bar.
HelpFile	**String** that identifies the Help file to use to provide context-sensitive Help for the dialog box. If HelpFile is provided, Context MUST also be provided.
Context	**Long** that is the Help context number assigned to the appropriate Help topic by the Help author. If Context is provided, HelpFile MUST also be provided.

Runtime Semantics.

- Displays a message in a dialog box, waits for the user to click a button, and returns an Integer indicating which button the user clicked.
- The Buttons argument settings are:

Constant	Value	Description
vbOKOnly	0	Display OK button only.
vbOKCancel	1	Display OK and Cancel buttons.
vbAbortRetryIgnore	2	Display Abort, Retry, and Ignore buttons.
vbYesNoCancel	3	Display Yes, No, and Cancel buttons.

Constant	Value	Description
vbYesNo	4	Display Yes and No buttons.
vbRetryCancel	5	Display Retry and Cancel buttons.
vbCritical	16	Display Critical Message icon.
vbQuestion	32	Display Warning Query icon.
vbExclamation	48	Display Warning Message icon.
vbInformation	64	Display Information Message icon.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user MUST respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right aligned
vbMsgBoxRtlReading	1048576	Specifies text SHOULD appear as right-to-left reading on Hebrew and Arabic systems

- The first group of values (05) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the buttons argument, use only one number from each group.

- The MsgBox function can return one of the following values:

Constant	Value	Description
vbOK	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIgnore	5	Ignore

Constant	Value	Description
vbYes	6	Yes
vbNo	7	No

- When both HelpFile and Context are provided, the user can press F1 to view the Help topic corresponding to the context. Some host applications, for example, Microsoft Excel 2010, also automatically add a Help button to the dialog box.
- If the dialog box displays a Cancel button, pressing the ESC key has the same effect as clicking Cancel. If the dialog box contains a Help button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.
- Note: to specify more than the first named argument, you MUST use MsgBox in an expression.

  To omit some positional arguments, you MUST include the corresponding comma delimiter.

6.1.2.8.1.14 Partition

Function Declaration

```
Function Partition(Number As Variant, Start As Variant, Stop As Variant, Interval As Variant)
As Variant
```

Parameter	Description
Number	**Long** to be evaluated against the ranges.
Start	**Long** that is the start of the overall range of numbers. The number can't be less than 0.
Stop	**Long** that is the end of the overall range of numbers. The number can't be equal to or less than Start.

Runtime Semantics.

- Returns a **String** indicating where a number occurs within a calculated series of ranges.
- The Partition function identifies the particular range in which Number falls and returns a **String** describing that range. The Partition function is most useful in queries. You can create a select query that shows how many orders fall within various ranges, for example, order values from 1 to 1000, 1001 to 2000, and so on.
- The following table shows how the ranges are determined using three sets of Start, Stop, and Interval parts. The First Range and Last Range columns show what Partition returns. The ranges are represented by lowervalue:uppervalue, where the low end (lowervalue) of the range is separated from the high end (uppervalue) of the range with a colon (:).

Start	Stop	Interval	Before First	First Range	Last Range	After Last
0	99	5	" :-1"	" 0: 4"	" 95: 99"	" 100: "
20	199	10	" : 19"	" 20: 29"	" 190: 199"	" 200: "

Start	Stop	Interval	Before First	First Range	Last Range	After Last
100	1010	20	" : 99"	" 100: 119"	" 1000: 1010"	" 1011: "

- In the preceding table, the third line shows the result when Start and Stop define a set of numbers that can't be evenly divided by Interval. The last range extends to Stop (11 numbers) even though Interval is 20.
- If necessary, Partition returns a range with enough leading spaces so that there are the same number of characters to the left and right of the colon as there are characters in Stop, plus one. This ensures that if you use Partition with other numbers, the resulting text will be handled properly during any subsequent sort operation.
- If Interval is 1, the range is number:number, regardless of the Start and Stop arguments. For example, if Interval is 1, Number is 100 and Stop is 1000, Partition returns " 100: 100".
- If any of the parts is Null, Partition returns the data value Null.

6.1.2.8.1.15 Shell

Function Declaration

```
Function Shell(PathName As Variant, Optional WindowStyle As VbAppWinStyle = vbMinimizedFocus)
As Double
```

Parameter	Description
PathName	**String**, containing the name of the program to execute and any required arguments or command-line switches; can include directory or folder and drive.
WindowStyle	**Integer** corresponding to the style of the window in which the program is to be run. If WindowStyle is omitted, the program is started minimized, with focus.

Runtime Semantics.

- Runs an executable program and returns a **Double** representing the implementation-defined program's task ID if successful, otherwise it returns the data value 0.
- The WindowStyle parameter accepts these values:

Constant	Value	Description
vbHide	0	Window is hidden and focus is passed to the hidden window.
vbNormalFocus	1	Window has focus and is restored to its original size and position.

Constant	Value	Description
vbMinimizedFocus	2	Window is displayed as an icon with focus.
vbMaximizedFocus	3	Window is maximized with focus.
vbNormalNoFocus	4	Window is restored to its most recent size and position. The currently active window remains active.
vbMinimizedNoFocus	6	Window is displayed as an icon. The currently active window remains active.

- If the Shell function successfully executes the named file, it returns the task ID of the started program. The task ID is an implementation-defined unique number that identifies the running program. If the Shell function can't start the named program, an error occurs.
- Note: by default, the Shell function runs other programs asynchronously. This means that a program started with Shell might not finish executing before the statements following the Shell function are executed.

6.1.2.8.1.16 Switch

Function Declaration

```
Function Switch(ParamArray VarExpr() As Variant) As Variant
```

Parameter	Description
VarExpr	Array of type **Variant** containing expressions to be evaluated.
Value	Value or expression to be returned if the corresponding expression is True.

Runtime Semantics.

- Evaluates a list of expressions and returns a **Variant** value or an expression associated with the first expression in the list that evaluates to the data value True.
- The Switch function argument list consists of pairs of expressions and values. The expressions are evaluated from left to right, and the value associated with the first expression to evaluate to True is returned. If the parts aren't properly paired, a run-time error occurs. For example, if VarExpr(0) evaluates to the data value True, Switch returns VarExpr(1). If VarExpr(0) evaluates to the data value False, but VarExpr(2) evaluates to the data value True, Switch returns VarExpr(3), and so on.
- Switch returns a Null value if:
  - None of the expressions evaluates to the data value True.
  - The first True expression has a corresponding value that is the data value Null.
- Switch evaluates all of the expressions, even though it returns only one of them. For example, if the evaluation of any expression results in a division by zero error, an error occurs.

6.1.2.8.2 Public Subroutines

6.1.2.8.2.1 AppActivate

Function Declaration

```
Sub AppActivate(Title As Variant, Optional Wait As Variant)
```

Parameter	Description
Title	**String** specifying the title in the title bar of the application window to activate. The task ID returned by the Shell function can be used in place of title to activate an application.
Wait	**Boolean** value specifying whether the calling application has the focus before activating another. If False (default), the specified application is immediately activated, even if the calling application does not have the focus. If True, the calling application waits until it has the focus, then activates the specified application.

Runtime Semantics.

- Activates an application window.
- The AppActivate statement changes the focus to the named application or window but does not affect whether it is maximized or minimized. Focus moves from the activated application window when the user takes some action to change the focus or close the window. Use the Shell function to start an application and set the window style.
- In determining which application to activate, Title is compared to the title string of each running application. If there is no exact match, any application whose title string begins with Title is activated. If there is more than one instance of the application named by Title, the window that is activated is implementation-defined.

6.1.2.8.2.2 Beep

Function Declaration

```
Sub Beep()
```

Runtime Semantics.

- Sounds a tone through the computer's speaker.
- The frequency and duration of the beep depend on hardware and system software, and vary among computers.

6.1.2.8.2.3 DeleteSetting

Function Declaration

```
Sub DeleteSetting(AppName As String, Optional Section As String, Optional Key As String)
```

Parameter	Description
AppName	**String** expression containing the name of the application or project to which the section or key setting applies.
Section	**String** expression containing the name of the section where the key setting is being deleted. If only AppName and Section are provided, the specified section is deleted along with all related key settings.
Key	**String** expression containing the name of the key setting being deleted.

Runtime Semantics.

- Deletes a section or key setting from an application's entry in an implementation dependent application registry.
- If all arguments are provided, the specified setting is deleted. A run-time error occurs if you attempt to use the DeleteSetting statement on a non-existent Section or Key setting.

6.1.2.8.2.4 SaveSetting

Function Declaration

```
Sub SaveSetting(AppName As String, Section As String, Key
As String, Setting As String)
```

Parameter	Description
AppName	**String** expression containing the name of the application or project to which the setting applies.
Section	**String** expression containing the name of the section where the key setting is being saved.
Key	**String** expression containing the name of the key setting being saved.
Setting	**String** expression containing the value that key is being set to.

Runtime Semantics.

- Saves or creates an application entry in the application's entry in the implementation dependent application registry.
- An error occurs if the key setting can’t be saved for any reason.

6.1.2.8.2.5 SendKeys

Function Declaration

```
Sub SendKeys(String As String, Optional Wait As Variant)
```

Parameter	Description
String	**String** expression specifying the keystrokes to send.
Wait	**Boolean** containing a value specifying the wait mode. If it evaluates to the data value False (default), control is returned to the procedure immediately after the keys are sent. If it evaluates to the data value **True**, keystrokes MUST be processed before control is returned to the procedure.

Runtime Semantics.

- Sends one or more keystrokes to the active window as if typed at the keyboard.
- Each key is represented by one or more characters. To specify a single keyboard character, use the character itself. For example, to represent the letter A, use

  "A"

  for String. To represent more than one character, append each additional character to the one preceding it. To represent the letters A, B, and C, use

  "ABC"

  for String.

  The plus sign (+), caret (^), percent sign (%), tilde (~), and parentheses ( ) have special meanings to SendKeys. To specify one of these characters, enclose it within braces (

  {}

). For example, to specify the plus sign, use

  {+}

  Brackets ([ ]) have no special meaning to SendKeys, but you MUST enclose them in braces. In other applications, brackets do have a special meaning that can be significant when dynamic data exchange (DDE) occurs. To specify brace characters, use

  {{}

  and

  {}}

  To specify characters that aren't displayed when you press a key, such as ENTER or TAB, and keys that represent actions rather than characters, use the codes shown in the following table:

Key	Code
BACKSPACE	{BACKSPACE}, {BS}, or {BKSP}
BREAK	{BREAK}
CAPS LOCK	{CAPSLOCK}
DEL or DELETE	{DELETE} or {DEL}
DOWN ARROW	{DOWN}

Key	Code
END	{END}
ENTER	{ENTER}or ~
ESC	{ESC}
HELP	{HELP}
HOME	{HOME}
INS or INSERT	{INSERT} or {INS}
LEFT ARROW	{LEFT}
NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
PRINT SCREEN	{PRTSC}
RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}
TAB	{TAB}
UP ARROW	{UP}
F1	{F1}
F2	{F2}
F3	{F3}
F4	{F4}
F5	{F5}
F6	{F6}
F7	{F7}
F8	{F8}
F9	{F9}
F10	{F10}
F11	{F11}
F12	{F12}
F13	{F13}

- To specify keys combined with any combination of the SHIFT, CTRL, and ALT keys, precede the key code with one or more of the following codes:

Key	Code
SHIFT	+
CTRL	^
ALT	%

- To specify that any combination of SHIFT, CTRL, and ALT SHOULD be held down while several other keys are pressed, enclose the code for those keys in parentheses. For example, to specify to hold down SHIFT while E and C are pressed, use "+(EC)". To specify to hold down SHIFT while E is pressed, followed by C without SHIFT, use "+EC".

- To specify repeating keys, use the form {key number}. You MUST put a space between key and number. For example, {LEFT 42} means press the LEFT ARROW key 42 times; {h 10} means press H 10 times.

6.1.2.9 KeyCodeConstants

Constant	Value	Description
vbKeyLButton	1	Left mouse button
vbKeyRButton	2	Right mouse button
vbKeyCancel	3	CANCEL key
vbKeyMButton	4	Middle mouse button
vbKeyBack	8	BACKSPACE key
vbKeyTab	9	TAB key
vbKeyClear	12	CLEAR key
vbKeyReturn	13	ENTER key
vbKeyShift	16	SHIFT key
vbKeyControl	17	CTRL key
vbKeyMenu	18	MENU key
vbKeyPause	19	PAUSE key
vbKeyCapital	20	CAPS LOCK key
vbKeyEscape	27	ESC key
vbKeySpace	32	SPACEBAR key
vbKeyPageUp	33	PAGE UP key
vbKeyPageDown	34	PAGE DOWN key
vbKeyEnd	35	END key
vbKeyHome	36	HOME key
vbKeyLeft	37	LEFT ARROW key
vbKeyUp	38	UP ARROW key
vbKeyRight	39	RIGHT ARROW key
vbKeyDown	40	DOWN ARROW key
vbKeySelect	41	SELECT key
vbKeyPrint	42	PRINT SCREEN key
vbKeyExecute	43	EXECUTE key
vbKeySnapshot	44	SNAPSHOT key

Constant	Value	Description
vbKeyInsert	45	INS key
vbKeyDelete	46	DEL key
vbKeyHelp	47	HELP key
vbKeyNumlock	144	NUM LOCK key
vbKeyA	65	A key
vbKeyB	66	B key
vbKeyC	67	C key
vbKeyD	68	D key
vbKeyE	69	E key
vbKeyF	70	F key
vbKeyG	71	G key
vbKeyH	72	H key
vbKeyI	73	I key
vbKeyJ	74	J key
vbKeyK	75	K key
vbKeyL	76	L key
vbKeyM	77	M key
vbKeyN	78	N key
vbKeyO	79	O key
vbKeyP	80	P key
vbKeyQ	81	Q key
vbKeyR	82	R key
vbKeyS	83	S key
vbKeyT	84	T key
vbKeyU	85	U key
vbKeyV	86	V key
vbKeyW	87	W key
vbKeyX	88	X key
vbKeyY	89	Y key
vbKeyZ	90	Z key
vbKey0	48	0 key
vbKey1	49	1 key
vbKey2	50	2 key
vbKey3	51	3 key
vbKey4	52	4 key

Constant	Value	Description
vbKey5	53	5 key
vbKey6	54	6 key
vbKey7	55	7 key
vbKey8	56	8 key
vbKey9	57	9 key
vbKeyNumpad0	96	Numpad 0 key
vbKeyNumpad1	97	Numpad 1 key
vbKeyNumpad2	98	Numpad 2 key
vbKeyNumpad3	99	Numpad 3 key
vbKeyNumpad4	100	Numpad 4 key
vbKeyNumpad5	101	Numpad 5 key
vbKeyNumpad6	102	Numpad 6 key
vbKeyNumpad7	103	Numpad 7 key
vbKeyNumpad8	104	Numpad 8 key
vbKeyNumpad9	105	Numpad 9 key
vbKeyMultiply	106	Numpad MULTIPLICATION SIGN (*) key
vbKeyAdd	107	Numpad PLUS SIGN (+) key
vbKeySeparator	108	Numpad ENTER (keypad) key
vbKeySubtract	109	Numpad MINUS SIGN (-) key
vbKeyDecimal	110	Numpad DECIMAL POINT(.) key
vbKeyDivide	111	Numpad DIVISION SIGN (/) key
vbKeyF1	112	F1 key
vbKeyF2	113	F2 key
vbKeyF3	114	F3 key
vbKeyF4	115	F4 key
vbKeyF5	116	F5 key
vbKeyF6	117	F6 key
vbKeyF7	118	F7 key
vbKeyF8	119	F8 key
vbKeyF9	120	F9 key
vbKeyF10	121	F10 key
vbKeyF11	122	F11 key
vbKeyF12	123	F12 key
vbKeyF13	124	F13 key
vbKeyF14	125	F14 key

Constant	Value	Description
vbKeyF15	126	F15 key
vbKeyF16	127	F16 key

6.1.2.10 Math

6.1.2.10.1 Public Functions

6.1.2.10.1.1 Abs

Function Declaration

```
Function Abs(Number As Variant) As Variant
```

Parameter	Description
Number	Any data value.

Runtime Semantics.

- If Number is the data value Null, returns Null.
- If Number is the data value Empty, returns the Integer data value 0.
- If Number is of a numeric value type, returns a value of the same value type specifying the absolute value of a number.
- Otherwise, the data value of Number is **Let**-coerced to **Double** and the absolute value of that data value is returned.

6.1.2.10.1.2 Atn

Function Declaration

```
Function Atn(Number As Double) As Double
```

Parameter	Description
Number	**Double** containing any valid numeric expression.

Runtime Semantics.

- Returns a **Double** specifying the arctangent of a number.
- The Atn function takes the ratio of two sides of a right triangle (Number) and returns the corresponding angle in radians. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

- The range of the result is -pi/2 to pi/2 radians.

6.1.2.10.1.3 Cos

Function Declaration

```
Function Cos(Number As Double) As Double
```

Parameter	Description
Number	**Double** containing any valid numeric expression that expresses an angle in radians.

Runtime Semantics.

- Returns a **Double** specifying the cosine of an angle.
- The Cos function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1.

6.1.2.10.1.4 Exp

Function Declaration

```
Function Exp(Number As Double) As Double
```

Parameter	Description
Number	**Double** containing any valid numeric expression.

Runtime Semantics.

- Returns a **Double** specifying e (the base of natural logarithms) raised to a power.
- If the value of Number exceeds 709.782712893, an error occurs. The constant e is approximately 2.718282.

6.1.2.10.1.5 Log

Function Declaration

```
Function Log(Number As Double) As Double
```

Parameter	Description
Number	**Double** containing any valid numeric expression greater than zero.

Runtime Semantics.

- Returns a **Double** specifying the natural logarithm of a number.
- The natural logarithm is the logarithm to the base e. The constant e is approximately 2.718282.

6.1.2.10.1.6 Rnd

Function Declaration

```
Function Rnd(Optional Number As Variant) As Single
```

Parameter	Description
Number	**Single** containing any valid numeric expression.

Runtime Semantics.

- Returns a **Single** containing a random number, according to the following table:

If number is	Rnd generates
Less than zero	The same number every time, using Number as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

- The Rnd function returns a value less than 1 but greater than or equal to zero.
- The value of Number determines how Rnd generates a random number:
    - For any given initial seed, the same number sequence is generated because each successive call to the Rnd function uses the previous number as a seed for the next number in the sequence.
- Before calling Rnd, use the Randomize statement without an argument to initialize the random-number generator with a seed based on the system timer. ☐ To produce random integers in a given range, use this formula:

  Int((upperbound - lowerbound + 1) * Rnd + lowerbound)

  Here, upperbound is the highest number in the range, and lowerbound is the lowest number in the range.
- An implementation is only required to repeat sequences of random numbers when Rnd is called with a negative argument before calling Randomize with a numeric argument. Using Randomize without calling Rnd in such a way yields implementation-defined results.

- The Rnd function necessarily generates numbers in a predictable sequence, and therefore is not required to use cryptographically-random number generators.

6.1.2.10.1.7 Round

Function Declaration

```
Function Round(Number As Variant, Optional
NumDigitsAfterDecimal As Long) As Variant
```

Parameter	Description
Number	**Variant** containing the numeric expression being rounded.
NumDigitsAfterDecimal	**Long** indicating how many places to the right of the decimal are included in the rounding. If omitted, integers are returned by the Round function.

Runtime Semantics.

- Returns a number rounded to a specified number of decimal places.

6.1.2.10.1.8 Sgn

Function Declaration

```
Function Sgn(Number As Variant) As Variant
```

Parameter	Description
Number	**Double** containing any valid numeric expression.

Runtime Semantics.

- Returns an **Integer** indicating the sign of a number, according to the following table:

If number is	Sgn returns
Greater than zero	1
Equal to zero	0
Less than zero	-1

- The sign of the number argument determines the return value of the Sgn function.

6.1.2.10.1.9 Sin

Function Declaration

```
Function Sin(Number As Double) As Double
```

Parameter	Description
Number	**Double** containing any valid numeric expression that expresses an angle in radians.

Runtime Semantics.

- Returns a **Double** specifying the sine of an angle.
- The Sin function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.
- The result lies in the range -1 to 1.

6.1.2.10.1.10Sqr

Function Declaration

```
Function Sqr(Number As Double) As Double
```

Parameter	Description
Number	**Double** containing any valid numeric expression greater than zero.

Runtime Semantics.

- Returns a **Double** specifying the square root of a number.

6.1.2.10.1.11Tan

Function Declaration

```
Function Tan(Number As Double) As Double
```

Parameter	Description
Number	**Double** containing any valid numeric expression that expresses an angle in radians.

Runtime Semantics.

- Returns a **Double** specifying the tangent of an angle.

- Tan takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

6.1.2.10.2 Public Subroutines

6.1.2.10.2.1 Randomize

Function Declaration

```
Sub Randomize(Optional Number As Variant)
```

Parameter	Description
Number	Empty or numeric seed value. If the argument is not Empty it MUST be **Let**-coercible to **Double**. Read Only

Runtime Semantics.

- Initializes the random-number generator.
- Randomize uses Number to initialize the Rnd function's random-number generator, giving it a new seed value. If the argument is missing or Empty, the value returned by the system timer is used as the new seed value.
- If Randomize is not used, the Rnd function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value.
- An implementation is only required to repeat sequences of random numbers when Rnd is called with a negative argument before calling Randomize with a numeric argument. Using Randomize without calling Rnd in such a way yields implementation-defined results.

6.1.2.11 Strings

6.1.2.11.1 Public Functions

6.1.2.11.1.1 Asc / AscW

Function Declaration

```
Function Asc(StringValue As String) As Integer
```

Parameter	Description
StringValue	**String** expression that SHOULD contain at least one character.

Runtime Semantics.

- Returns an **Integer** data value representing the 7-bit ASCII code point of the first character of StringValue. If the character does not correspond to an ASCII character the result is implementation defined.
- Code point value greater than 32,767 are returned as negative **Integer** data values.
- If the argument is the null string ("") Error Number 5 ("Invalid procedure call or argument") is raised.

6.1.2.11.1.2 AscB

Function Declaration

```
Function AscB(StringValue As String) As Integer
```

Parameter	Description
StringValue	**String** expression that SHOULD contain at least one character.

Runtime Semantics.

- Returns an **Integer** data value that is the first eight bits (the first byte) of the implementation dependent character encoding of the string. If individual character code points more than 8 bits it is implementation dependent as to whether the bits returned are the high order or low order bits of the code point.
- If the argument is the null string ("") Error Number 5 ("Invalid procedure call or argument") is raised.

6.1.2.11.1.3 AscW

Function Declaration

```
Function AscW(StringValue As String) As Integer
```

Parameter	Description
StringValue	**String** expression that SHOULD contain at least one character.

Runtime Semantics.

- If the implemented uses 16-bit **Unicode** code points returns an **Integer** data value that is the 16-bit Unicode code point of the first character of StringValue.
- If the implementation does not support Unicode, return the result of Asc(StringValue).
- Code point values greater than 32,767 are returned as negative **Integer** data values.

- If the argument is the null string ("") Error Number 5 ("Invalid procedure call or argument") is raised.

6.1.2.11.1.4 Chr / Chr$

Function Declaration

```
Function Chr(CharCode As Long) As Variant
Function Chr$(CharCode As Long) As String
```

Parameter	Description
CharCode	**Long** whose value is a code point.

Runtime Semantics.

- Returns a **String** data value consisting of a single character containing the character whose code point is the data value of the argument.
- If the argument is not in the range 0 to 255, Error Number 5 ("Invalid procedure call or argument") is raised unless the implementation supports a character set with a larger code point range.
- If the argument value is in the range of 0 to 127, it is interpreted as a 7-bit ASCII code point.
- If the argument value is in the range of 128 to 255, the code point interpretation of the value is implementation defined.
- Chr$ has the same runtime semantics as Chr, however the declared type of its function result is **String** rather than **Variant**.

6.1.2.11.1.5 ChrB / ChrB$

Function Declaration

```
Function ChrB(CharCode As Long) As Variant
Function ChrB$(CharCode As Long) As String
```

Parameter	Description
CharCode	**Long** whose value is a code point.

Runtime Semantics.

- Returns a **String** data value consisting of a single byte character whose code point value is the data value of the argument.
- If the argument is not in the range 0 to 255, Error Number 6 ("Overflow") is raised.
- ChrB$ has the same runtime semantics as ChrB however the declared type of its function result is **String** rather than **Variant**.

- Note: the ChrB function is used with byte data contained in a String. Instead of returning a character, which can be one or two bytes, ChrB returns a single byte. The ChrW function returns a String containing the **Unicode** character except on platforms where Unicode is not supported, in which case, the behavior is identical to the Chr function.

6.1.2.11.1.6 ChrW/ ChrW$

Function Declaration

```
Function ChrW(CharCode As Long) As Variant
Function ChrW$(CharCode As Long) As String
```

Parameter	Description
CharCode	**Long** whose value is a code point.

Runtime Semantics.

- Returns a **String** data value consisting of a single character containing the character whose code point is the data value of the argument.
- If the argument is not in the range -32,767 to 65,535 then Error Number 5 ("Invalid procedure call or argument") is raised.
- If the argument is a negative value it is treated as if it was the value: CharCode + 65,536.
- If the implemented uses 16-bit **Unicode** code points argument, data value is interpreted as a 16-bit Unicode code point.
- If the implementation does not support Unicode, ChrW has the same semantics as Chr.
- ChrW$ has the same runtime semantics as ChrW, however the declared type of its function result is **String** rather than **Variant**.

6.1.2.11.1.7 Filter

Function Declaration

```
Function Filter(SourceArray() As Variant, Match As String,
Optional Include As Boolean = True, Optional Compare As
VbCompareMethod = vbBinaryCompare)
```

Parameter	Description
SourceArray	**Variant** containing one-dimensional array of strings to be searched.
Match	**String** to search for.

Parameter	Description
Include	**Boolean** value indicating whether to return substrings that include or exclude match. If include is True, Filter returns the subset of the array that contains match as a substring. If include is False, Filter returns the subset of the array that does not contain match as a substring.
Compare	Numeric value indicating the kind of string comparison to use. See the next table in this section for values.

Runtime Semantics.

- Returns a zero-based array containing subset of a string array based on a specified filter criteria.
- The Compare argument can have the following values (if omitted, it uses the <option-compare-directive> of the calling module):

Constant	Value	Description
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.

- If no matches of Match are found within SourceArray, Filter returns an empty array. An error occurs if SourceArray is the data value Null or is not a one-dimensional array.
- The array returned by the Filter function contains only enough elements to contain the number of matched items.

6.1.2.11.1.8 Format

Function Declaration

```
Function Format(Expression As Variant, Optional Format As
Variant, Optional FirstDayOfWeek As VbDayOfWeek = vbSunday,
Optional FirstWeekOfYear As VbFirstWeekOfYear = vbFirstJan1)
```

Parameter	Description
Expression	Any valid expression.
Format	A valid named or user-defined format expression.
FirstDayOfWeek	A constant that specifies the first day of the week.
FirstWeekOfYear	A constant that specifies the first week of the year.

Runtime Semantics.

- Returns a **String** containing an expression formatted according to instructions contained in a format expression.

- The FirstDayOfWeek argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use NLS API setting.
VbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

- The FirstWeekOfYear argument has these settings:

Constant	Value	Description
vbUseSystem	0	Use NLS API setting.
vbFirstJan1	1	Start with week in which January 1 occurs (default).
vbFirstFourDays	2	Start with the first week that has at least four days in the year.
vbFirstFullWeek	3	Start with the first full week of the year.

- To determine how to format a certain type of data, see the following table:

To Format	Do This
Numbers	Use predefined named numeric formats or create user-defined numeric formats.
Dates and times	Use predefined named date/time formats or create user-defined date/time formats.
Date and time serial numbers	Use date and time formats or numeric formats.
Strings	Create a user-defined string format.

- If you try to format a number without specifying Format, Format provides functionality similar to the Str function, although it is internationally aware. However, positive numbers formatted as strings using Format do not include a leading space reserved for the sign of the value; those converted using Str retain the leading space.

- When formatting a non-localized numeric string, use a user-defined numeric format to ensure that it gets formatted correctly.

- Note: if the Calendar property setting is Gregorian and format specifies date formatting, the supplied expression MUST be Gregorian. If the Visual Basic Calendar property setting is Hijri, the supplied expression MUST be Hijri.
- If the calendar is Gregorian, the meaning of format expression symbols is unchanged. If the calendar is Hijri, all date format symbols (for example, dddd, mmmm, yyyy) have the same meaning but apply to the Hijri calendar. Format symbols remain in English; symbols that result in text display (for example, AM and PM) display the string (English or Arabic) associated with that symbol. The range of certain symbols changes when the calendar is Hijri.

Symbol	Range
d	1-30
dd	1-30
ww	1-51
mmm	Displays full month names (Hijri month names have no abbreviations).
y	1-355
yyyy	100-9666

6.1.2.11.1.9 Format$

This function is functionally identical to the Format function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.10 FormatCurrency

Function Declaration

```
Function FormatCurrency(Expression As Variant, Optional
NumDigitsAfterDecimal As Long = -1, Optional
IncludeLeadingDigit As VbTriState = vbUseDefault, Optional
UseParensForNegativeNumbers As VbTriState = vbUseDefault,
Optional GroupDigits As VbTriState = vbUseDefault) As
String
```

Parameter	Description
Expression	**Variant** containing the expression to be formatted.
NumDigitsAfterDecimal	Numeric value indicating how many places to the right of the decimal are displayed. Default value is 1, which indicates that the computer's *regional settings* are used.
IncludeLeadingDigit	Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See the next table in this section for values.

Parameter	Description
UseParensForNegativeNumbers	Tristate constant that indicates whether or not to place negative values within parentheses. See the next table in this section for values.
GroupDigits	Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's *regional settings*. See the next table in this section for values.

Runtime Semantics.

- Returns an expression formatted as a currency value using the implementation-defined currency symbol.
- The IncludeLeadingDigit, UseParensForNegativeNumbers, and GroupDigits arguments have the following settings:

Constant	Value	Description
vbTrue	1	True
vbFalse	0	False
vbUseDefault	2	Implementation-defined value.

- Returns an expression formatted as a currency value using the implementation-defined currency symbol.
- When one or more optional arguments are omitted, the values for omitted arguments are implementation-defined.
- The position of the currency symbol relative to the currency value is implementation-defined.

6.1.2.11.1.11 FormatDateTime

Function Declaration

```
Function FormatDateTime(Expression As Variant, NamedFormat As VbDateTimeFormat =
vbGeneralDate) As String
```

Parameter	Description
Date	**Variant** containing a **Date** expression to be formatted.
NamedFormat	Numeric value that indicates the date/time format used. If omitted, vbGeneralDate is used.

Runtime Semantics.

- Returns an expression formatted as a date or time.
- The NamedFormat argument has the following settings:

Constant	Value	Description
vbGeneralDate	0	Display a date and/or time. If there is a date part, display it as a short date. If there is a time part, display it as a long time. If present, both parts are displayed.
vbLongDate	1	Display a date using the implementation-defined long date format.
vbShortDate	2	Display a date using the implementation-defined short date format.
vbLongTime	3	Display a time using the implementation-defined time format.
vbShortTime	4	Display a time using the 24-hour format (hh:mm).

6.1.2.11.1.12FormatNumber

Function Declaration

```
Function FormatNumber(Expression, Optional
NumDigitsAfterDecimal As Long = -1, Optional
IncludeLeadingDigit As VbTriState = vbUseDefault, Optional
UseParensForNegativeNumbers As VbTriState = vbUseDefault,
Optional GroupDigits As VbTriState = vbUseDefault) As String
```

Parameter	Description
Expression	**Variant** containing the expression to be formatted.
NumDigitsAfterDecimal	Numeric value indicating how many places to the right of the decimal are displayed. Default value is 1, which indicates that implementation-defined settings are used.
IncludeLeadingDigit	Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See the next table in this section for values.
UseParensForNegativeNumbers	Tristate constant that indicates whether or not to place negative values within parentheses. See the next table in this section for values.
GroupDigits	Tristate constant that indicates whether or not numbers are grouped using the implementation-defined group delimiter. See the next table in this section for values.

Runtime Semantics.

- Returns an expression formatted as a number.
- The IncludeLeadingDigit, UseParensForNegativeNumbers, and GroupDigits arguments have the following settings:

Constant	Value	Description
vbTrue	1	True
vbFalse	0	False
vbUseDefault	2	Implementation-defined value.

- Returns an expression formatted as a number.
- When one or more optional arguments are omitted, the values for omitted arguments are provided by the computer's *regional settings*.

6.1.2.11.1.13 FormatPercent

Function Declaration

```
Function FormatPercent(Expression, Optional
NumDigitsAfterDecimal As Long = -1, Optional
IncludeLeadingDigit As VbTriState = vbUseDefault, Optional
UseParensForNegativeNumbers As VbTriState = vbUseDefault,
Optional GroupDigits As VbTriState = vbUseDefault) As String
```

Parameter	Description
Expression	**Variant** containing the expression to be formatted.
NumDigitsAfterDecimal	Numeric value indicating how many places to the right of the decimal are displayed. Default value is 1, which indicates that implementation-defined settings are used.
IncludeLeadingDigit	Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See the next table in this section for values.
UseParensForNegativeNumbers	Tristate constant that indicates whether or not to place negative values within parentheses. See the next table in this section for values.
GroupDigits	Tristate constant that indicates whether or not numbers are grouped using the implementation-defined group delimiter. See the next table in this section for values.

Runtime Semantics.

- Returns an expression formatted as a percentage (multiplied by 100) with a trailing % character.

- The IncludeLeadingDigit, UseParensForNegativeNumbers, and GroupDigits arguments have the following settings:

Constant	Value	Description
vbTrue	1	True
vbFalse	0	False
vbUseDefault	2	Use the setting from the computer's *regional settings*.

- When one or more optional arguments are omitted, the values for omitted arguments are implementation-defined.

6.1.2.11.1.14InStr / InStrB

Function Declaration

```
Function InStr(Optional Arg1 As Variant, Optional Arg2 As
Variant, Optional Arg3 As Variant, Optional Compare As
VbCompareMethod = vbBinaryCompare)
```

If Arg3 is not present then Arg1 is used as the string to be searched, and Arg2 is used as the pattern (and the start position is 1). If Arg3 IS present then Arg1 is used as a string and Arg2 is used as the pattern.

Parameter	Description
Arg1	Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. If start contains the data value Null, an error occurs. This argument is required if Compare is specified.
Arg2	String expression to search.
Arg3	String expression sought.
Compare	Specifies the type of string comparison. If compare is the data value Null, an error occurs. If Compare is omitted, the Option Compare setting determines the type of comparison. Specify a valid LCID (LocaleID) to use locale-specific rules in the comparison.

Runtime Semantics.

- Returns a **Long** specifying the position of the first occurrence of one string within another.
- The Compare argument can have the following values (if omitted, it uses the <option-compare-directive> of the calling module):

Constant	Value	Description
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.

- InStr returns the following values:

If	InStr returns
Arg2 is zero-length	0
Arg2 is Null	Null
Arg3 is zero-length	Arg1
Arg3 is Null	Null
Arg3 is not found	0
Arg3 is found within Arg2	Position at which match is found
Arg1 > Arg3	0

- The InStrB function is used with byte data contained in a string. Instead of returning the character position of the first occurrence of one string within another, InStrB returns the byte position.

6.1.2.11.1.15InStrRev

Function Declaration

```
Function InStrRev(StringCheck As String, StringMatch As
String, Optional Start As Long = -1, Optional Compare As VbCompareMethod = vbBinaryCompare)
As Long
```

Parameter	Description
StringCheck	**String** expression to search.
StringMatch	**String** expression being searched for.
Start	**Long** containing a numeric expression that sets the starting position for each search. If omitted, the data value 1 is used, which means that the search begins at the last character position. If Start contains the data value Null, an error occurs.
Compare	Numeric value indicating the kind of comparison to use when evaluating substrings. If omitted, a binary comparison is performed. See the next table in this section for values.

Runtime Semantics.

- Returns the position of an occurrence of one string within another, from the end of string.

- The Compare argument can have the following values (if omitted, it uses the <option-compare-directive> of the calling module):

Constant	Value	Description
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.

- InStrRev returns the following values:

If	InStrRev returns
StringCheck is zero-length	0
StringCheck is Null	Null
StringMatch is zero-length	Start
StringMatch is Null	Null
StringMatch is not found	0
StringMatch is found within StringCheck	Position at which match is found
Start > Len(StringMatch)	0

6.1.2.11.1.16Join

Function Declaration

```
Function Join(SourceArray() As Variant, Optional Delimiter As Variant) As String
```

Parameter	Description
SourceArray	**Variant** containing one-dimensional array containing substrings to be joined.
Delimiter	String character used to separate the substrings in the returned string. If omitted, the space character (" ") is used. If Delimiter is a zero-length string (""), all items in the list are concatenated with no delimiters.

Runtime Semantics.

- Returns a string created by joining a number of substrings contained in an array.

6.1.2.11.1.17LCase

Function Declaration

```
Function LCase(String As Variant)
```

Parameter	Description
String	**Variant** containing any valid **String** expression. If String contains the data value Null, Null is returned.

Runtime Semantics.

- Returns a String that has been converted to lowercase.
- Only uppercase letters are converted to lowercase; all lowercase letters and non-letter characters remain unchanged.

6.1.2.11.1.18 LCase$

This function is functionally identical to the LCase function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.19 Left / LeftB

Function Declaration

```
Function Left(String, Length As Long)
```

Parameter	Description
String	String expression from which the leftmost characters are returned. If string contains Null, Null is returned.
Length	**Long** containing a Numeric expression indicating how many characters to return. If it equals the data value 0, a zero-length string ("") is returned. If it's greater than or equal to the number of characters in String, the entire string is returned.

Runtime Semantics.

- Returns a **String** containing a specified number of characters from the left side of a string.
- Note: use the LeftB function with byte data contained in a string. Instead of specifying the number of characters to return, length specifies the number of bytes.

6.1.2.11.1.20 Left$

This function is functionally identical to the Left function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.21 LeftB$

This function is functionally identical to the LeftB function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.22Len / LenB

Function Declaration

```
Function Len(Expression As Variant) As Variant
Function LenB(Expression As Variant) As Variant
```

Parameter	Description
Expression	Any valid string expression, or any valid variable name. If the variable name is a **Variant**, Len/LenB treats it the same as a **String** and always returns the number of characters it contains.

Runtime Semantics.

- Returns a **Long** containing the number of characters in a string or the number of bytes required to store a variable on the current platform.
- If Expression contains the data value Null, Null is returned.
- With user-defined types, Len returns the size as it will be written to the file.
- LenB will return the same value as Len, except for strings or UDTs:
  - LenB can return different values than Len for **Unicode** strings or double-byte character set (DBCS) representations. Instead of returning the number of characters in a string, LenB returns the number of bytes used to represent that string.
  - With user-defined types, LenB returns the in-memory size, including any implementation-specific padding between elements.
- Note: Len might not be able to determine the actual number of storage bytes required when used with variable-length strings in user-defined data types.

6.1.2.11.1.23LTrim / RTrim / Trim

Function Declaration

```
Function LTrim(String As Variant) As Variant
Function RTrim(String As Variant) As Variant
Function Trim(String As Variant) As Variant
```

Parameter	Description
String	**Variant**, containing any valid **String** expression.

Runtime Semantics.

- Returns a **String** containing a copy of a specified string without leading spaces (LTrim), trailing spaces (RTrim), or both leading and trailing spaces (Trim).
- If String contains the data value Null, Null is returned.

6.1.2.11.1.24 LTrim$ / RTrim$ / Trim$

These functions are functionally identical to the LTrim, RTrim, and Trim functions respectively, with the exception that the return type of these functions is **String** rather than **Variant**.

6.1.2.11.1.25 Mid / MidB

Function Declaration

```
Function Mid(String As Variant, Start As Long, Optional
Length As Variant) As Variant
```

Parameter	Description
String	**String** expression from which characters are returned. If String contains the data value Null, Null is returned.
Start	**Long** containing the character position in String at which the part to be taken begins. If Start is greater than the number of characters in String, Mid returns a zero-length string ("").
Length	**Long** containing the number of characters to return. If omitted or if there are fewer than Length characters in the text (including the character at start), all characters from the start position to the end of the string are returned.

Runtime Semantics.

- Returns a **String** containing a specified number of characters from a string.
- To determine the number of characters in String, use the Len function.
- Note: use the MidB function with byte data contained in a string, as in double-byte character set languages. Instead of specifying the number of characters, the arguments specify numbers of bytes.

6.1.2.11.1.26 Mid$

This function is functionally identical to the Mid function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.27 MidB$

This function is functionally identical to the MidB function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.28 MonthName

Function Declaration

```
Function MonthName(Month As Long, Optional Abbreviate As Boolean = False) As String
```

Parameter	Description
Month	**Long** containing the numeric designation of the month. For example, January is 1, February is 2, and so on.
Abbreviate	**Boolean** value that indicates if the month name is to be abbreviated. If omitted, the default is False, which means that the month name is not abbreviated.

Runtime Semantics.

- Returns a **String** indicating the specified month.

6.1.2.11.1.29 Replace

Function Declaration

```
Function Replace(Expression As String, Find As String,
Replace As String, Optional Start As Long = 1, Optional Count As Long = -1, Optional Compare
As VbCompareMethod = vbBinaryCompare) As String
```

Parameter	Description
Expression	**String** expression containing substring to replace.
Find	Substring being searched for.
Replace	Replacement substring.
Start	Position within expression where substring search is to begin. If omitted, the data value 1 is assumed.
Count	Number of substring substitutions to perform. If omitted, the default value is the data value 1, which means make all possible substitutions.
Compare	Numeric value indicating the kind of comparison to use when evaluating substrings. See the next table in this section for values.

Runtime Semantics.

- Returns a **String** in which a specified substring has been replaced with another substring a specified number of times.
- The Compare argument can have the following values (if omitted, it uses the <option-compare-directive> of the calling module):

Constant	Value	Description
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.

- Replace returns the following values:

If	Replace returns
Expression is zero-length	Zero-length string ("")
Expression is Null	An error.
Find is zero-length	Copy of Expression.
Replace is zero-length	Copy of Expression with all occurrences of Find removed.
Start > Len(Expression)	Zero-length string.
Count is 0	Copy of Expression.

- The return value of the Replace function is a **String**, with substitutions made, that begins at the position specified by Start and concludes at the end of the Expression string. It is not a copy of the original string from start to finish.

6.1.2.11.1.30 Right / RightB

Function Declaration

```
Function Right(String, Length As Long)
```

Parameter	Description
String	**String** expression from which the rightmost characters are returned. If string contains the data value Null, Null is returned.
Length	**Long** containing the numeric expression indicating how many characters to return. If it equals the data value 0, a zero-length string ("") is returned. If it is greater than or equal to the number of characters in String, the entire string is returned.

Runtime Semantics.

- Returns a **String** containing a specified number of characters from the right side of a string.
- To determine the number of characters in string, use the Len function.

- Note: use the RightB function with byte data contained in a **String**. Instead of specifying the number of characters to return, length specifies the number of bytes.

6.1.2.11.1.31 Right$

This function is functionally identical to the Right function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.32 RightB$

This function is functionally identical to the RightB function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.33 Space

Function Declaration

```
Function Space(Number As Long) As Variant
```

Parameter	Description
Number	**Long** containing the number of spaces in the **String**.

Runtime Semantics.

- Returns a **String** consisting of the specified number of spaces.
- The Space function is useful for formatting output and clearing data in fixed-length strings.

6.1.2.11.1.34 Space$

This function is functionally identical to the Space function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.35 Split

Function Declaration

```
Function Split(Expression As String, Optional Delimiter As
Variant, Optional Limit As Long = -1, Optional Compare As VbCompareMethod = vbBinaryCompare)
```

Parameter	Description
Expression	**String** expression containing substrings and delimiters. If expression is a zero-length string(""), Split returns an empty array, that is, an array with no elements and no data.
Delimiter	**String** containing the character used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If delimiter is a zero-length string, a single-element array containing the entire expression string is returned.

Parameter	Description
Limit	Number of substrings to be returned; the data value 1 indicates that all substrings are returned.
Compare	Numeric value indicating the kind of comparison to use when evaluating substrings. See the next table in this section for values.

Runtime Semantics.

- Returns a zero-based, one-dimensional array containing a specified number of substrings.
- The Compare argument can have the following values (if omitted, it uses the <option-compare-directive> of the calling module):

Constant	Value	Description
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.

6.1.2.11.1.36StrComp

Function Declaration

```
Function StrComp(String1 As Variant, String2 As Variant,
Optional Compare As VbCompareMethod = vbBinaryCompare)
```

Parameter	Description
String1	Any valid **String** expression.
String2	Any valid **String** expression.
Compare	Specifies the type of string comparison. If the Compare argument is the data value Null, an error occurs.

Runtime Semantics.

- Returns an **Integer** indicating the result of a string comparison.
- The Compare argument can have the following values (if omitted, it uses the <option-compare-directive> of the calling module):

Constant	Value	Description
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.

- The StrComp function has the following return values:

If	StrComp returns
String1 is less than String2	-1
String1 is equal to String2	0
String1 is greater than String2	1
String1 or String2 is Null	Null

6.1.2.11.1.37StrConv

Function Declaration

```
Function StrConv(String As Variant, Conversion As VbStrConv, LocaleID As Long) As Variant
```

Parameter	Description
String	**String** containing the expression to be converted.
Conversion	**Integer** containing the sum of values specifying the type of conversion to perform.
LCID	The LocaleID, if different than the default implementation-defined LocaleID.

Runtime Semantics.

- Returns a **String** converted as specified.
- The Conversion argument settings are:

Constant	Value	Description
vbUpperCase	1	Converts the string to uppercase characters.
vbLowerCase	2	Converts the string to lowercase characters.
vbProperCase	3	Converts the first letter of every word in string to uppercase.
vbWide*	4*	Converts narrow (single-byte) characters in string to wide (double-byte) characters.
vbNarrow*	8*	Converts wide (double-byte) characters in string to narrow (single-byte) characters.
vbKatakana**	16**	Converts Hiragana characters in string to Katakana characters.
vbHiragana**	32**	Converts Katakana characters in string to Hiragana characters.

Constant	Value	Description
vbUnicode	64	Converts the string to **Unicode** using the default **code page** of the system.
vbFromUnicode	128	Converts the string from Unicode to the default code page of the system.

*Applies to East Asia locales.

**Applies to Japan only.

- Note: these constants are specified by VBA, and as a result, they can be used anywhere in code in place of the actual values. Most can be combined, for example, vbUpperCase + vbWide, except when they are mutually exclusive, for example, vbUnicode + vbFromUnicode. The constants vbWide, vbNarrow, vbKatakana, and vbHiragana cause run-time errors when used in locales where they do not apply.

- The following are valid word separators for proper casing: Null (Chr$(0)), horizontal tab

  (Chr$(9)), linefeed (Chr$(10)), vertical tab (Chr$(11)), form feed (Chr$(12)), carriage return (Chr$(13)), space (SBCS) (Chr$(32)). The actual value for a space varies by country/region for DBCS.

- When converting from a **Byte** array in ANSI format to a **String**, use the StrConv function. When converting from such an array in Unicode format, use an assignment statement.

6.1.2.11.1.38String

Function Declaration

```
Function String(Number As Long, Character As Variant) As
Variant
```

Parameter	Description
Number	Long specifying the length of the returned string. If number contains the data value Null, Null is returned.
Character	**Variant** containing the character code specifying the character or string expression whose first character is used to build the return string. If character contains Null, Null is returned.

Runtime Semantics.

- Returns a **String** containing a repeating character string of the length specified.

- If Character is a number greater than 255, String converts the number to a valid character code using the formula: character Mod 256

6.1.2.11.1.39String$

This function is functionally identical to the String function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.40StrReverse

Function Declaration

```
Function StrReverse(Expression As String) As String
```

Parameter	Description
Expression	**String** whose characters are to be reversed.

Runtime Semantics.

- Returns a **String** in which the character order of a specified **String** is reversed.
- If Expression is a zero-length string (""), a zero-length string is returned. If Expression is Null, an error occurs.

6.1.2.11.1.41UCase

Function Declaration

```
Function UCase(String As Variant)
```

Parameter	Description
String	**Variant** containing any valid **String** expression. If String contains the data value Null, Null is returned.

Runtime Semantics.

- Returns a **String** that has been converted to uppercase.
- Only lowercase letters are converted to uppercase; all uppercase letters and non-letter characters remain unchanged.

6.1.2.11.1.42UCase$

This function is functionally identical to the UCase function, with the exception that the return type of the function is **String** rather than **Variant**.

6.1.2.11.1.43WeekdayName

Function Declaration

```
Function WeekdayName(Weekday As Long, Optional Abbreviate
As Boolean = False, Optional FirstDayOfWeek As VbDayOfWeek
= vbUseSystemDayOfWeek) As String
```

Parameter	Description
Weekday	**Long** containing the numeric designation for the day of the week. Numeric value of each day depends on setting of the FirstDayOfWeek setting.
Abbreviate	**Boolean** value that indicates if the weekday name is to be abbreviated. If omitted, the default is False, which means that the weekday name is not abbreviated.
FirstDayOfWeek	Numeric value indicating the first day of the week. See the next table in this section for values.

Runtime Semantics.

- Returns a **String** indicating the specified day of the week.
- The FirstDayOfWeek argument can have the following values:

Constant	Value	Description
vbUseSystem	0	Use National Language Support (NLS) API setting.
vbSunday	1	Sunday (default)
vbMonday	2	Monday
vbTuesday	3	Tuesday
vbWednesday	4	Wednesday
vbThursday	5	Thursday
vbFriday	6	Friday
vbSaturday	7	Saturday

6.1.2.12 SystemColorConstants

Whenever their values are used in contexts expecting a color value, these system color constants SHOULD be interpreted as their specified implementation-dependent colors.

Constant	Value	Description
vbScrollBars	&H80000000	Scroll bar color
vbDesktop	&H80000001	Desktop color
vbActiveTitleBar	&H80000002	Color of the title bar for the active window
vbInactiveTitleBar	&H80000003	Color of the title bar for the inactive window
vbMenuBar	&H80000004	Menu background color

Constant	Value	Description
vbWindowBackground	&H80000005	Window background color
vbWindowFrame	&H80000006	Window frame color
vbMenuText	&H80000007	Color of text on menus
vbWindowText	&H80000008	Color of text in windows
vbTitleBarText	&H80000009	Color of text in caption, size box, and scroll arrow
vbActiveBorder	&H8000000A	Border color of active window
vbInactiveBorder	&H8000000B	Border color of inactive window
vbApplicationWorkspace	&H8000000C	Background color of multiple-document interface (MDI) applications
vbHighlight	&H8000000D	Background color of items selected in a control
vbHighlightText	&H8000000E	Text color of items selected in a control
vbButtonFace	&H8000000F	Color of shading on the face of command buttons
vbButtonShadow	&H80000010	Color of shading on the edge of command buttons
vbGrayText	&H80000011	Grayed (disabled) text
vbButtonText	&H80000012	Text color on push buttons
vbInactiveCaptionText	&H80000013	Color of text in an inactive caption
vb3DHighlight	&H80000014	Highlight color for 3D display elements
vb3DDKShadow	&H80000015	Darkest shadow color for 3D display elements
vb3DLight	&H80000016	Second lightest of the 3D colors after **vb3Dhighlight**
vb3DFace	&H8000000F	Color of text face
vb3Dshadow	&H80000010	Color of text shadow
vbInfoText	&H80000017	Color of text in ToolTips
vbInfoBackground	&H80000018	Background color of ToolTips

6.1.3 Predefined Class Modules

6.1.3.1 Collection Object

The Collection class defines the behavior of a collection, which represents a sequence of values.

6.1.3.1.1 Public Functions

6.1.3.1.1.1 Count

Function Declaration

```
Function Count() As Long
```

Runtime Semantics.

- Returns the number of objects in a collection.

6.1.3.1.1.2 Item

Function Declaration

```
Function Item(Index As Variant) As Variant
```

Parameter	Description
Index	An expression that specifies the position of a member of the collection. If a numeric expression, Index MUST be a number from 1 to the value of the collection's Count property. If a string expression, Index MUST correspond to the Key argument specified when the member referred to was added to the collection.

Runtime Semantics.

- Returns a specific member of a Collection object either by position or by key.
- If the value provided as Index does not match any existing member of the collection, an error occurs.
- The Item method is the default method for a collection. Therefore, the following lines of code are equivalent:

  Print MyCollection(1)

  Print MyCollection.Item(1)

6.1.3.1.2 Public Subroutines

6.1.3.1.2.1 Add

Function Declaration

```
Sub Add(Item As Variant, Optional Key As Variant, Optional Before As Variant, Optional After
As Variant)
```

Parameter	Description
Item	An expression of any type that specifies the member to add to the collection.
Key	A unique **String** expression that specifies a key string that can be used, instead of a positional index, to access a member of the collection.

Parameter	Description
Before	An expression that specifies a relative position in the collection. The member to be added is placed in the collection before the member identified by the before argument. If a numeric expression, before MUST be a number from 1 to the value of the collection's Count property. If a **String** expression, before MUST correspond to the key specified when the member being referred to was added to the collection. Either a Before position or an After position can be specified, but not both.
After	An expression that specifies a relative position in the collection. The member to be added is placed in the collection after the member identified by the After argument. If numeric, After MUST be a number from 1 to the value of the collection's Count property. If a **String**, After MUST correspond to the Key specified when the member referred to was added to the collection. Either a Before position or an After position can be specified, but not both.

Runtime Semantics.

- Adds a member to a Collection object.
- Whether the before or after argument is a string expression or numeric expression, it MUST refer to an existing member of the collection, or an error occurs.
- An error also occurs if a specified Key duplicates the key for an existing member of the collection.
- An implementation can define a maximum number of elements that a Collection object can contain.

6.1.3.1.2.2 Remove

Function Declaration

```
Sub Remove(Index As Variant)
```

Parameter	Description
Index	An expression that specifies the position of a member of the collection. If a numeric expression, Index MUST be a number from 1 to the value of the collection's Count property. If a **String** expression, Index MUST correspond to the Key argument specified when the member referred to was added to the collection.

Runtime Semantics.

- Removes a member from a Collection object.
- If the value provided as Index doesn’t match an existing member of the collection, an error occurs.

6.1.3.2 Err Class

The Err Class defines the behavior of its sole instance, known as the *Err object*. *The Err object’s* properties and methods reflect and control the error state of the active *VBA Environment* and can be

accessed inside any procedure. The Err Class is a *global class module (section 5.2.4.1.2)* with a *default instance variable (section 5.2.4.1.2)* so its sole instance can be directly referenced using the name Err.

6.1.3.2.1 Public Subroutines

6.1.3.2.1.1 Clear

Function Declaration

```
Sub Clear()
```

Runtime Semantics.

- Clears all property settings of the Err object.
- The Clear method is called automatically whenever any of the following statements is executed:
  - Resume statement (section 5.4.4.2)
  - Exit Sub (section 5.4.2.17)
  - Exit Function (section 5.4.2.18)
  - Exit Property (section 5.4.2.19)
  - On Error statement (section 5.4.4.1)

6.1.3.2.1.2 Raise

Function Declaration

```
Sub Raise(Number As Long, Optional Source As Variant,
Optional Description As Variant, Optional HelpFile As Variant, Optional HelpContext As
Variant)
```

Parameter	Description
Number	**Long** that identifies the nature of the error. VBA errors (both VBA-defined and user-defined errors) are in the range 0-65535. The range 0-512 is reserved for system errors; the range 513-65535 is available for user-defined errors. When setting the Number property to a custom error code in a class module, add the error code number to the vbObjectError constant. For example, to generate the error number 513, assign vbObjectError + 513 to the Number property.
Source	**String** expression naming the object or application that generated the error. When setting this property for an object, use the form project.class. If Source is not specified, current *project name (section 4.1)* is used.

Parameter	Description
Description	**String** expression describing the error. If unspecified, the value in Number is examined. If it can be mapped to a VBA run-time error code, the **String** that would be returned by the Error function is used as Description. If there is no VBA error corresponding to Number, the "Application-defined or object-defined error" message is used.
HelpFile	The fully qualified path to the Help file in which help on this error can be found. If unspecified, this value is implementation-defined.
HelpContext	The context ID identifying a topic within HelpFile that provides help for the error. If omitted, this value is implementation-defined.

Runtime Semantics.

- Generates a run-time error.
- If Raise is invoked without specifying some arguments, and the property settings of the Err object contain values that have not been cleared, those values serve as the values for the new error.
- Raise is used for generating run-time errors and can be used instead of the Error statement (section 5.4.4.3). Raise is useful for generating errors when writing class modules, because the Err object gives richer information than possible when generating errors with the Error statement. For example, with the Raise method, the source that generated the error can be specified in the Source property, online Help for the error can be referenced, and so on.

6.1.3.2.2 Public Properties

6.1.3.2.2.1 Description

6.1.3.2.2.2 HelpContext

```
Property HelpContext As Long
```

Runtime Semantics.

- Returns or sets a **String** expression containing the context ID for a topic in a Help file.
- The HelpContext property is used to automatically display the Help topic specified in the HelpFile property. If both HelpFile and HelpContext are empty, the value of Number is checked. If Number corresponds to a VBA run-time error value, then the implementation-defined VBA Help context ID for the error is used. If the Number value doesn't correspond to a VBA error, an implementation-defined Help screen is displayed.

6.1.3.2.2.3 HelpFile

```
Property HelpFile As String
```

Runtime Semantics.

- Returns or sets a **String** expression containing the fully qualified path to a Help file.
- If a Help file is specified in HelpFile, it is automatically called when the user presses the Help button (or the F1 KEY) in the error message dialog box. If the HelpContext property contains a

valid context ID for the specified file, that topic is automatically displayed. If no HelpFile is specified, an implementation-defined Help file is displayed.

6.1.3.2.2.4 LastDIIError

```
Property LastDllError As Long
```

Runtime Semantics.

- Returns a system error code produced by a call to a dynamic-link library (DLL). This value is read-only.
- The LastDLLError property applies only to DLL calls made from VBA code. When such a call is made, the called function usually returns a code indicating success or failure, and the LastDLLError property is filled. Check the documentation for the DLL's functions to determine the return values that indicate success or failure. Whenever the failure code is returned, the VBA application SHOULD immediately check the LastDLLError property. No error is raised when the LastDLLError property is set.

6.1.3.2.2.5 Number

```
Property Number As Long
```

Runtime Semantics.

- Returns or sets a numeric value specifying an error. Number is the Err object's default property.
- When returning a user-defined error from an object, set Err.Number by adding the number selected as an error code to the vbObjectError constant. For example, use the following code to return the number 1051 as an error code:

    Err.Raise Number := vbObjectError + 1051, Source:= "SomeClass"

6.1.3.2.2.6 Source

```
Property Source As String
```

Runtime Semantics.

- Returns or sets a **String** expression specifying the name of the object or application that originally generated the error.
- This property has an implementation-defined default value for errors raised within VBA code.

6.1.3.3 Global Class

6.1.3.3.1 Public Subroutines

6.1.3.3.1.1 Load

Subroutine Declaration

```
Sub Load(Object As Object)
```

Runtime Semantics.

- Loads a form or control into memory.
- Using the Load statement with forms is unnecessary unless you want to load a form without displaying it. Any reference to a form (except in a Set or If...TypeOf statement) automatically loads it if it's not already loaded. For example, the Show method loads a form before displaying it. Once the form is loaded, its properties and controls can be altered by the application, whether or not the form is actually visible.
- When VBA loads a Form object, it sets form properties to their initial values and then performs the Load event procedure. When an application starts, VBA automatically loads and displays the application's startup form.
- When loading a Form whose MDIChild property is set to True (in other words, the child form) before loading an MDIForm, the MDIForm is automatically loaded before the child form. MDI child forms cannot be hidden, and thus are immediately visible after the Form_Load event procedure ends.

6.1.3.3.1.2 Unload

Unloads a form or control from memory.

Subroutine Declaration

```
Sub Unload(Object As Object)
```

Runtime Semantics.

- Unloads a form or control into memory.
- Unloading a form or control can be necessary or expedient in some cases where the memory used is needed for something else, or when there is a need to reset properties to their original values.
- Before a form is unloaded, the Query_Unload event procedure occurs, followed by the Form_Unload event procedure. Setting the cancel argument to True in either of these events prevents the form from being unloaded. For MDIForm objects, the MDIForm object's Query_Unload event procedure occurs, followed by the Query_Unload event procedure and Form_Unload event procedure for each MDI child form, and finally the MDIForm object's Form_Unload event procedure.
- When a form is unloaded, all controls placed on the form at run time are no longer accessible. Controls placed on the form at design time remain intact; however, any run-time changes to those controls and their properties are lost when the form is reloaded. All changes to form properties are also lost. Accessing any controls on the form causes it to be reloaded.
- Note: when a form is unloaded, only the displayed component is unloaded. The code associated with the form module remains in memory.
- Only control array elements added to a form at run time can be unloaded with the Unload statement. The properties of unloaded controls are reinitialized when the controls are reloaded.

7 Change Tracking

This section identifies changes that were made to this document since the last release. Changes are classified as Major, Minor, or None.

The revision class **Major** means that the technical content in the document was significantly revised. Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements.
- A document revision that captures changes to protocol functionality.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **None** means that no new technical changes were introduced. Minor editorial and formatting changes may have been made, but the relevant technical content is identical to the last released version.

The changes made to this document are listed in the following table. For more information, please contact dochelp@microsoft.com.

Section	Description	Revision class
3.3.2 Number Tokens	Two rules were moved to separate lines.	Minor
3.3.2 Number Tokens	Updated the rule name.	Minor
3.3.3 Date Tokens	Added hyphen for time-value and date-value.	Minor
3.3.3 Date Tokens	Separated rules English-month-name and English-month-abbreviation into two lines.	Minor
3.3.5 Identifier Tokens	Two rules were moved to separate lines.	Minor
3.3.5 Identifier Tokens	Updated DIGIT to decimal-digit.	Minor
3.3.5.1 Non-Latin Identifiers	Removed space from Korean-identifier-character.	Minor
3.3.5.3 Special Identifier Forms	Two rules were moved to separate lines.	Minor
3.4 Conditional Compilation	Two rules were moved to separate lines.	Minor
3.4 Conditional Compilation	Added hyphen to <conditional-module-body>.	Minor
3.4.1 Conditional Compilation Const Directive	Two rules were moved to separate lines.	Minor
3.4.2 Conditional Compilation If Directives	Added hyphen for cc-else-block element.	Minor
5.2 Module Declaration Section Structure	Added hyphens for public-external-procedure-declaration and common-option-directive.	Minor
5.2.1.2 Option Base Directive	Updated <base-directive> to <option-base-directive>.	Minor

Section	Description	Revision class
5.2.2 Implicit Definition Directives	Multiple rules were moved to separate lines.	Minor
5.2.3.2 Const Declarations	Added hyphen for <const-as-clause>.	Minor
5.2.3.2 Const Declarations	Two rules were moved to separate lines.	Minor
5.2.3.3 User Defined Type Declarations	Multiple rules were moved to separate lines.	Minor
5.2.3.3 User Defined Type Declarations	Updated <module> to module.	Minor
5.2.3.4 Enum Declarations	Updated the rule name.	Minor
5.2.3.4 Enum Declarations	Removed the statement enum-declaration = public-enum-declaration / private-enum-declaration.	Minor
5.2.3.4 Enum Declarations	Updated <enum-member-name> to <enum-member>.	Minor
5.3.1 Procedure Declarations	Two rules were moved to separate lines.	Minor
5.3.1.3 Procedure Names	Two rules were moved to separate lines.	Minor
5.3.1.5 Parameter Lists	Updated <dim-statement> and <-const-statement> to <local-variable-declaration>, <static-variable-declaration> and <local-const-declaration>.	Minor
5.3.1.5 Parameter Lists	Updated <def-statement> to <def-directive>.	Minor
5.3.1.9 Implemented Name Declarations	Updated the element name.	Minor
5.4.2.3 For Statement	Added a hyphen	Minor
5.4.2.4 For Each Statement	Updated <nested-forstatement> to <nested-for-statement>	Minor
5.4.2.8 If Statement	Added hyphen for the boolean-expression.	Minor
5.4.2.9 Single-line If Statement	Updated <single-line-else> to <single-line-else-clause>.	Minor
5.4.2.10 Select Case Statement	Updated <case-else-block> to <case-else-clause>.	Minor
5.4.2.16 On…GoSub Statement	Updated <procedure-block> to <procedure-body>.	Minor
5.4.2.19 Exit Property Statement	Updated <exit-procedure-statement> to <exit-property-statement>.	Minor
5.4.3.8 Let Statement	Added hyphen for <lexpression>.	Minor
5.4.5.8.1 Output Lists	Two rules were moved to separate lines.	Minor
5.4.5.11 Put Statement	Updated <record-length> to <rec-length>	Minor

Section	Description	Revision class
5.6.9 Operator Expressions	Updated the names for the operator rules.	Minor
5.6.9.3 Arithmetic Operators	Updated the names for the operator rules.	Minor
5.6.9.4 & Operator	Updated the name for the operator rule.	Minor
5.6.9.5 Relational Operators	Added hyphen for greaterthan-operator.	Minor
5.6.9.5 Relational Operators	Updated the names for the operator rules.	Minor
5.6.9.6 Like Operator	Updated the name for the operator rule.	Minor
5.6.9.7 Is Operator	Updated the name for the operator rule.	Minor
5.6.9.8 Logical Operators	Added hyphen for impoperator.	Minor
5.6.9.8 Logical Operators	Updated the names for the operator rules.	Minor

8 Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X