Data Types in Visual Basic

Visual Studio 2015

The *data type* of a programming element refers to what kind of data it can hold and how it stores that data. Data types apply to all values that can be stored in computer memory or participate in the evaluation of an expression. Every variable, literal, constant, enumeration, property, procedure parameter, procedure argument, and procedure return value has a data type.

Declared Data Types

You define a programming element with a declaration statement, and you specify its data type with the **As** clause. The following table shows the statements you use to declare various elements.

Programming element	Data type declaration

Variable	In a Dim Statement (Visual Basic)
	`Dim amount As Double`
	`Static yourName As String`
	`Public billsPaid As Decimal = 0`

Literal	With a literal type character; see "Literal Type Characters" in Type Characters (Visual Basic) `Dim searchChar As Char = "."C`
Constant	In a Const Statement (Visual Basic) `Const modulus As Single = 4.17825F`
Enumeration	In an Enum Statement (Visual Basic) `Public Enum colors`
Property	In a Property Statement `Property region() As String`
Procedure parameter	In a Sub Statement (Visual Basic), Function Statement (Visual Basic), or Operator Statement `Sub addSale(ByVal amount As Double)`

Procedure argument	In the calling code; each argument is a programming element that has already been declared, or an expression containing declared elements `subString = Left(inputString, 5)`
Procedure return value	In a Function Statement (Visual Basic) or Operator Statement `Function convert(ByVal b As Byte) As String`

For a list of Visual Basic data types, see Data Type Summary (Visual Basic).

See Also

Type Characters (Visual Basic)
Elementary Data Types (Visual Basic)
Composite Data Types (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
Value Types and Reference Types
Type Conversions in Visual Basic
Structures (Visual Basic)
Troubleshooting Data Types (Visual Basic)
Data Type Summary (Visual Basic)
Efficient Use of Data Types (Visual Basic)

Troubleshooting Data Types (Visual Basic)

Visual Studio 2015

This page lists some common problems that can occur when you perform operations on intrinsic data types.

Floating-Point Expressions Do Not Compare as Equal

When you work with floating-point numbers (Single Data Type (Visual Basic) and Double Data Type (Visual Basic)), remember that they are stored as binary fractions. This means they cannot hold an exact representation of any quantity that is not a binary fraction (of the form *k / (2 ^ n)* where *k* and *n* are integers). For example, 0.5 (= 1/2) and 0.3125 (= 5/16) can be held as precise values, whereas 0.2 (= 1/5) and 0.3 (= 3/10) can be only approximations.

Because of this imprecision, you cannot rely on exact results when you operate on floating-point values. In particular, two values that are theoretically equal might have slightly different representations.

To compare floating-point quantities

1. Calculate the absolute value of their difference by using the Abs method of the Math class in the System namespace.
2. Determine an acceptable maximum difference, such that you can consider the two quantities to be equal for practical purposes if their difference is no larger.
3. Compare the absolute value of the difference to the acceptable difference.

The following example demonstrates both incorrect and correct comparison of two **Double** values.

VB

```
Dim oneThird As Double = 1.0 / 3.0
Dim pointThrees As Double = 0.333333333333333

' The following comparison does not indicate equality.
Dim exactlyEqual As Boolean = (oneThird = pointThrees)

' The following comparison indicates equality.
Dim closeEnough As Double = 0.000000000000001
Dim absoluteDifference As Double = Math.Abs(oneThird - pointThrees)
Dim practicallyEqual As Boolean = (absoluteDifference < closeEnough)

MsgBox("1.0 / 3.0 is represented as " & oneThird.ToString("G17") &
    vbCrLf & "0.333333333333333 is represented as " &
    pointThrees.ToString("G17") &
    vbCrLf & "Exact comparison generates " & CStr(exactlyEqual) &
    vbCrLf & "Acceptable difference comparison generates " &
```

```
        CStr(practicallyEqual))
```

The previous example uses the ToString method of the Double structure so that it can specify better precision than the **CStr** keyword uses. The default is 15 digits, but the "G17" format extends it to 17 digits.

Mod Operator Does Not Return Accurate Result

Because of the imprecision of floating-point storage, the Mod Operator (Visual Basic) can return an unexpected result when at least one of the operands is floating-point.

The Decimal Data Type (Visual Basic) does not use floating-point representation. Many numbers that are inexact in **Single** and **Double** are exact in **Decimal** (for example 0.2 and 0.3). Although arithmetic is slower in **Decimal** than in floating-point, it might be worth the performance decrease to achieve better precision.

To find the integer remainder of floating-point quantities

1. Declare variables as **Decimal**.
2. Use the literal type character **D** to force literals to **Decimal**, in case their values are too large for the **Long** data type.

The following example demonstrates the potential imprecision of floating-point operands.

VB

```
Dim two As Double = 2.0
Dim zeroPointTwo As Double = 0.2
Dim quotient As Double = two / zeroPointTwo
Dim doubleRemainder As Double = two Mod zeroPointTwo

MsgBox("2.0 is represented as " & two.ToString("G17") &
    vbCrLf & "0.2 is represented as " & zeroPointTwo.ToString("G17") &
    vbCrLf & "2.0 / 0.2 generates " & quotient.ToString("G17") &
    vbCrLf & "2.0 Mod 0.2 generates " &
    doubleRemainder.ToString("G17"))

Dim decimalRemainder As Decimal = 2D Mod 0.2D
MsgBox("2.0D Mod 0.2D generates " & CStr(decimalRemainder))
```

The previous example uses the ToString method of the Double structure so that it can specify better precision than the **CStr** keyword uses. The default is 15 digits, but the "G17" format extends it to 17 digits.

Because `zeroPointTwo` is **Double**, its value for 0.2 is an infinitely repeating binary fraction with a stored value of 0.20000000000000001. Dividing 2.0 by this quantity yields 9.9999999999999995 with a remainder of 0.19999999999999991.

In the expression for `decimalRemainder`, the literal type character `D` forces both operands to **Decimal**, and 0.2 has a precise representation. Therefore the **Mod** operator yields the expected remainder of 0.0.

Note that it is not sufficient to declare `decimalRemainder` as **Decimal**. You must also force the literals to **Decimal**, or they use **Double** by default and `decimalRemainder` receives the same inaccurate value as `doubleRemainder`.

Boolean Type Does Not Convert to Numeric Type Accurately

Boolean Data Type (Visual Basic) values are not stored as numbers, and the stored values are not intended to be equivalent to numbers. For compatibility with earlier versions, Visual Basic provides conversion keywords (CType Function (Visual Basic), **CBool**, **CInt**, and so on) to convert between **Boolean** and numeric types. However, other languages sometimes perform these conversions differently, as do the .NET Framework methods.

You should never write code that relies on equivalent numeric values for **True** and **False**. Whenever possible, you should restrict usage of **Boolean** variables to the logical values for which they are designed. If you must mix **Boolean** and numeric values, make sure that you understand the conversion method that you select.

Conversion in Visual Basic

When you use the **CType** or **CBool** conversion keywords to convert numeric data types to **Boolean**, 0 becomes **False** and all other values become **True**. When you convert **Boolean** values to numeric types by using the conversion keywords, **False** becomes 0 and **True** becomes -1.

Conversion in the Framework

The ToInt32 method of the Convert class in the System namespace converts **True** to +1.

If you must convert a **Boolean** value to a numeric data type, be careful about which conversion method you use.

Character Literal Generates Compiler Error

In the absence of any type characters, Visual Basic assumes default data types for literals. The default type for a character literal — enclosed in quotation marks (**" "**) — is **String**.

The **String** data type does not widen to the Char Data Type (Visual Basic). This means that if you want to assign a literal to a **Char** variable, you must either make a narrowing conversion or force the literal to the **Char** type.

To create a Char literal to assign to a variable or constant
1. Declare the variable or constant as **Char**. 2. Enclose the character value in quotation marks (**" "**). 3. Follow the closing double quotation mark with the literal type character **C** to force the literal to **Char**. This is necessary if the type checking switch (Option Strict Statement) is **On**, and it is desirable in any case.

The following example demonstrates both unsuccessful and successful assignments of a literal to a **Char** variable.

VB

```
Dim charVar As Char
' The following statement attempts to convert a String literal to Char.
' Because Option Strict is On, it generates a compiler error.
charVar = "Z"
' The following statement succeeds because it specifies a Char literal.
charVar = "Z"c
' The following statement succeeds because it converts String to Char.
charVar = CChar("Z")
```

There is always a risk in using narrowing conversions, because they can fail at run time. For example, a conversion from **String** to **Char** can fail if the **String** value contains more than one character. Therefore, it is better programming to use the **C** type character.

String Conversion Fails at Run Time

The String Data Type (Visual Basic) participates in very few widening conversions. **String** widens only to itself and **Object**, and only **Char** and **Char()** (a **Char** array) widen to **String**. This is because **String** variables and constants can contain values that other data types cannot contain.

When the type checking switch (Option Strict Statement) is **On**, the compiler disallows all implicit narrowing conversions. This includes those involving **String**. Your code can still use conversion keywords such as **CStr** and CType Function (Visual Basic), which direct the .NET Framework to attempt the conversion.

Note

The narrowing-conversion error is suppressed for conversions from the elements in a **For Each…Next** collection to the loop control variable. For more information and examples, see the "Narrowing Conversions" section in For Each...Next Statement (Visual Basic).

Narrowing Conversion Protection

The disadvantage of narrowing conversions is that they can fail at run time. For example, if a **String** variable contains anything other than "True" or "False," it cannot be converted to **Boolean**. If it contains punctuation characters, conversion to any numeric type fails. Unless you know that your **String** variable always holds values that the destination type can accept, you should not try a conversion.

If you must convert from **String** to another data type, the safest procedure is to enclose the attempted conversion in the Try...Catch...Finally Statement (Visual Basic). This lets you deal with a run-time failure.

Character Arrays

A single **Char** and an array of **Char** elements both widen to **String**. However, **String** does not widen to **Char()**. To convert a **String** value to a **Char** array, you can use the ToCharArray method of the System.String class.

Meaningless Values

In general, **String** values are not meaningful in other data types, and conversion is highly artificial and dangerous. Whenever possible, you should restrict usage of **String** variables to the character sequences for which they are designed. You should never write code that relies on equivalent values in other types.

See Also

Data Types in Visual Basic
Type Characters (Visual Basic)
Value Types and Reference Types
Type Conversions in Visual Basic
Data Type Summary (Visual Basic)
Type Conversion Functions (Visual Basic)
Efficient Use of Data Types (Visual Basic)

Data Type Summary (Visual Basic)

Visual Studio 2015

The following table shows the Visual Basic data types, their supporting common language runtime types, their nominal storage allocation, and their value ranges.

Visual Basic type	**Common language runtime type structure**	**Nominal storage allocation**	**Value range**
Boolean	Boolean	Depends on implementing platform	**True** or **False**
Byte	Byte	1 byte	0 through 255 (unsigned)
Char (single character)	Char	2 bytes	0 through 65535 (unsigned)
Date	DateTime	8 bytes	0:00:00 (midnight) on January 1, 0001 through 11:59:59 PM on December 31, 9999
Decimal	Decimal	16 bytes	0 through +/-79,228,162,514,264,337,593,543,950,335 (+/-7.9...E+28) † with no decimal point; 0 through +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/-0.0000000000000000000000000001 (+/-1E-28) †
Double (double-precision floating-point)	Double	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324 † for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 † for positive values
Integer	Int32	4 bytes	-2,147,483,648 through 2,147,483,647 (signed)
Long (long integer)	Int64	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (9.2...E+18 †) (signed)
Object	Object (class)	4 bytes on	Any type can be stored in a variable of type **Object**

		32-bit platform 8 bytes on 64-bit platform	
SByte	SByte	1 byte	-128 through 127 (signed)
Short (short integer)	Int16	2 bytes	-32,768 through 32,767 (signed)
Single (single-precision floating-point)	Single	4 bytes	-3.4028235E+38 through -1.401298E-45 † for negative values; 1.401298E-45 through 3.4028235E+38 † for positive values
String (variable-length)	String (class)	Depends on implementing platform	0 to approximately 2 billion Unicode characters
UInteger	UInt32	4 bytes	0 through 4,294,967,295 (unsigned)
ULong	UInt64	8 bytes	0 through 18,446,744,073,709,551,615 (1.8...E+19 †) (unsigned)
User-Defined (structure)	(inherits from ValueType)	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members
UShort	UInt16	2 bytes	0 through 65,535 (unsigned)

† In *scientific notation*, "E" refers to a power of 10. So 3.56E+2 signifies *3.56 x 10*^{2} or 356, and 3.56E-2 signifies *3.56 / 10*^{2} or 0.0356.

Note

For strings containing text, use the StrConv function to convert from one text format to another.

In addition to specifying a data type in a declaration statement, you can force the data type of some programming elements by using a type character. See Type Characters (Visual Basic).

Memory Consumption

When you declare an elementary data type, it is not safe to assume that its memory consumption is the same as its nominal storage allocation. This is due to the following considerations:

- **Storage Assignment.** The common language runtime can assign storage based on the current characteristics of the platform on which your application is executing. If memory is nearly full, it might pack your declared elements as closely together as possible. In other cases it might align their memory addresses to natural hardware boundaries to optimize performance.

- **Platform Width.** Storage assignment on a 64-bit platform is different from assignment on a 32-bit platform.

Composite Data Types

The same considerations apply to each member of a composite data type, such as a structure or an array. You cannot rely on simply adding together the nominal storage allocations of the type's members. Furthermore, there are other considerations, such as the following:

- **Overhead.** Some composite types have additional memory requirements. For example, an array uses extra memory for the array itself and also for each dimension. On a 32-bit platform, this overhead is currently 12 bytes plus 8 bytes for each dimension. On a 64-bit platform this requirement is doubled.
- **Storage Layout.** You cannot safely assume that the order of storage in memory is the same as your order of declaration. You cannot even make assumptions about byte alignment, such as a 2-byte or 4-byte boundary. If you are defining a class or structure and you need to control the storage layout of its members, you can apply the StructLayoutAttribute attribute to the class or structure.

Object Overhead

An **Object** referring to any elementary or composite data type uses 4 bytes in addition to the data contained in the data type.

See Also

StrConv
StructLayoutAttribute
Type Conversion Functions (Visual Basic)
Conversion Summary (Visual Basic)
Type Characters (Visual Basic)
Efficient Use of Data Types (Visual Basic)

Numeric Data Types (Visual Basic)

Visual Studio 2015

Visual Basic supplies several *numeric data types* for handling numbers in various representations. *Integral* types represent only whole numbers (positive, negative, and zero), and *nonintegral* types represent numbers with both integer and fractional parts.

For a table showing a side-by-side comparison of the Visual Basic data types, see Data Type Summary (Visual Basic).

Integral Numeric Types

Integral data types are those that represent only numbers without fractional parts.

The *signed* integral data types are SByte Data Type (Visual Basic) (8-bit), Short Data Type (Visual Basic) (16-bit), Integer Data Type (Visual Basic) (32-bit), and Long Data Type (Visual Basic) (64-bit). If a variable always stores integers rather than fractional numbers, declare it as one of these types.

The *unsigned* integral types are Byte Data Type (Visual Basic) (8-bit), UShort Data Type (Visual Basic) (16-bit), UInteger Data Type (32-bit), and ULong Data Type (Visual Basic) (64-bit). If a variable contains binary data, or data of unknown nature, declare it as one of these types.

Performance

Arithmetic operations are faster with integral types than with other data types. They are fastest with the **Integer** and **UInteger** types in Visual Basic.

Large Integers

If you need to hold an integer larger than the **Integer** data type can hold, you can use the **Long** data type instead. **Long** variables can hold numbers from -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807. Operations with **Long** are slightly slower than with **Integer**.

If you need even larger values, you can use the Decimal Data Type (Visual Basic). You can hold numbers from -79,228,162,514,264,337,593,543,950,335 through 79,228,162,514,264,337,593,543,950,335 in a **Decimal** variable if you do not use any decimal places. However, operations with **Decimal** numbers are considerably slower than with any other numeric data type.

Small Integers

If you do not need the full range of the **Integer** data type, you can use the **Short** data type, which can hold integers from -32,768 through 32,767. For the smallest integer range, the **SByte** data type holds integers from -128 through 127. If you have a very large number of variables that hold small integers, the common language runtime can sometimes store your **Short** and **SByte** variables more efficiently and save memory consumption. However, operations with **Short** and **SByte** are somewhat slower than with **Integer**.

Unsigned Integers

If you know that your variable never needs to hold a negative number, you can use the *unsigned types* **Byte**, **UShort**, **UInteger**, and **ULong**. Each of these data types can hold a positive integer twice as large as its corresponding signed type (**SByte**, **Short**, **Integer**, and **Long**). In terms of performance, each unsigned type is exactly as efficient as its corresponding signed type. In particular, **UInteger** shares with **Integer** the distinction of being the most efficient of all the elementary numeric data types.

Nonintegral Numeric Types

Nonintegral data types are those that represent numbers with both integer and fractional parts.

The nonintegral numeric data types are **Decimal** (128-bit fixed point), Single Data Type (Visual Basic) (32-bit floating point), and Double Data Type (Visual Basic) (64-bit floating point). They are all signed types. If a variable can contain a fraction, declare it as one of these types.

Decimal is not a floating-point data type. **Decimal** numbers have a binary integer value and an integer scaling factor that specifies what portion of the value is a decimal fraction.

You can use **Decimal** variables for money values. The advantage is the precision of the values. The **Double** data type is faster and requires less memory, but it is subject to rounding errors. The **Decimal** data type retains complete accuracy to 28 decimal places.

Floating-point (**Single** and **Double**) numbers have larger ranges than **Decimal** numbers but can be subject to rounding errors. Floating-point types support fewer significant digits than **Decimal** but can represent values of greater magnitude.

Nonintegral number values can be expressed as *mmmEeee*, in which *mmm* is the *mantissa* (the significant digits) and *eee* is the *exponent* (a power of 10). The highest positive values of the nonintegral types are 7.9228162514264337593543950335E+28 for **Decimal**, 3.4028235E+38 for **Single**, and 1.79769313486231570E+308 for **Double**.

Performance

Double is the most efficient of the fractional data types, because the processors on current platforms perform floating-point operations in double precision. However, operations with **Double** are not as fast as with the integral types such as **Integer**.

Small Magnitudes

For numbers with the smallest possible magnitude (closest to 0), **Double** variables can hold numbers as small as

-4.94065645841246544E-324 for negative values and 4.94065645841246544E-324 for positive values.

Small Fractional Numbers

If you do not need the full range of the **Double** data type, you can use the **Single** data type, which can hold floating-point numbers from -3.4028235E+38 through 3.4028235E+38. The smallest magnitudes for **Single** variables are -1.401298E-45 for negative values and 1.401298E-45 for positive values. If you have a very large number of variables that hold small floating-point numbers, the common language runtime can sometimes store your **Single** variables more efficiently and save memory consumption.

See Also

Elementary Data Types (Visual Basic)
Character Data Types (Visual Basic)
Miscellaneous Data Types (Visual Basic)
Troubleshooting Data Types (Visual Basic)
How to: Call a Windows Function that Takes Unsigned Types (Visual Basic)

Character Data Types (Visual Basic)

Visual Studio 2015

Visual Basic provides *character data types* to deal with printable and displayable characters. While they both deal with Unicode characters, **Char** holds a single character whereas **String** contains an indefinite number of characters.

For a table that displays a side-by-side comparison of the Visual Basic data types, see Data Type Summary (Visual Basic).

Char Type

The **Char** data type is a single two-byte (16-bit) Unicode character. If a variable always stores exactly one character, declare it as **Char**. For example:

VB

```
' Initialize the prefix variable to the character 'a'.
Dim prefix As Char = "a"
```

Each possible value in a **Char** or **String** variable is a *code point*, or character code, in the Unicode character set. Unicode characters include the basic ASCII character set, various other alphabet letters, accents, currency symbols, fractions, diacritics, and mathematical and technical symbols.

Note

The Unicode character set reserves the code points D800 through DFFF (55296 through 55551 decimal) for *surrogate pairs*, which require two 16-bit values to represent a single code point. A **Char** variable cannot hold a surrogate pair, and a **String** uses two positions to hold such a pair.

For more information, see Char Data Type (Visual Basic).

String Type

The **String** data type is a sequence of zero or more two-byte (16-bit) Unicode characters. If a variable can contain an indefinite number of characters, declare it as **String**. For example:

VB

```
' Initialize the name variable to "Monday".
Dim name As String = "Monday"
```

For more information, see String Data Type (Visual Basic).

See Also

Elementary Data Types (Visual Basic)
Composite Data Types (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
Value Types and Reference Types
Type Conversions in Visual Basic
Troubleshooting Data Types (Visual Basic)
Type Characters (Visual Basic)

Miscellaneous Data Types (Visual Basic)

Visual Studio 2015

Visual Basic supplies several data types that are not oriented toward numbers or characters. Instead, they deal with specialized data such as yes/no values, date/time values, and object addresses.

For a table showing a side-by-side comparison of the Visual Basic data types, see Data Type Summary (Visual Basic).

Boolean Type

The Boolean Data Type (Visual Basic) is an unsigned value that is interpreted as either **True** or **False**. Its data width depends on the implementing platform. If a variable can contain only two-state values such as true/false, yes/no, or on/off, declare it as **Boolean**.

Date Type

The Date Data Type (Visual Basic) is a 64-bit value that holds both date and time information. Each increment represents 100 nanoseconds of elapsed time since the beginning (12:00 AM) of January 1 of the year 1 in the Gregorian calendar. If a variable can contain a date value, a time value, or both, declare it as **Date**.

Object Type

The Object Data Type is a 32-bit address that points to an object instance within your application or in some other application. An **Object** variable can refer to any object your application recognizes, or to data of any data type. This includes both *value types*, such as **Integer**, **Boolean**, and structure instances, and *reference types*, which are instances of objects created from classes such as **String** and Form, and array instances.

If a variable stores a pointer to an instance of a class that you do not know at compile time, or if it can point to data of various data types, declare it as **Object**.

The advantage of the **Object** data type is that you can use it to store data of any data type. The disadvantage is that you incur extra operations that take more execution time and make your application perform slower. If you use an **Object** variable for value types, you incur *boxing* and *unboxing*. If you use it for reference types, you incur *late binding*.

See Also

Type Characters (Visual Basic)
Elementary Data Types (Visual Basic)
Numeric Data Types (Visual Basic)
Character Data Types (Visual Basic)
Troubleshooting Data Types (Visual Basic)

Early and Late Binding (Visual Basic)

Conversion Summary (Visual Basic)

Visual Studio 2015

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Convert ANSI value to string.	Chr, ChrW
Convert string to lowercase or uppercase.	Format, LCase,UCase
Convert date to serial number.	DateSerial, DateValue
Convert decimal number to other bases.	Hex, Oct
Convert number to string.	Format, Str
Convert one data type to another.	CBool, CByte, CDate, CDbl, CDec, CInt, CLng, CSng, CShort, CStr, CType, Fix, Int
Convert date to day, month, weekday, or year.	Day, Month, Weekday, Year
Convert time to hour, minute, or second.	Hour, Minute, Second
Convert string to ASCII value.	Asc, AscW
Convert string to number.	Val
Convert time to serial number.	TimeSerial, TimeValue

See Also

Keywords (Visual Basic)
Visual Basic Runtime Library Members

Composite Data Types (Visual Basic)

Visual Studio 2015

In addition to the elementary data types Visual Basic supplies, you can also assemble items of different types to create *composite data types* such as structures, arrays, and classes. You can build composite data types from elementary types and from other composite types. For example, you can define an array of structure elements, or a structure with array members.

Data Types

A composite type is different from the data type of any of its components. For example, an array of **Integer** elements is not of the **Integer** data type.

An array data type is normally represented using the element type, parentheses, and commas as necessary. For example, a one-dimensional array of **String** elements is represented as **String()**, and a two-dimensional array of **Boolean** elements is represented as **Boolean(,)**.

Structure Types

There is no single data type comprising all structures. Instead, each definition of a structure represents a unique data type, even if two structures define identical elements in the same order. However, if you create two or more instances of the same structure, Visual Basic considers them to be of the same data type.

Array Types

There is no single data type comprising all arrays. The data type of a particular instance of an array is determined by the following:

- The fact of being an array
- The rank (number of dimensions) of the array
- The element type of the array

In particular, the length of a given dimension is not part of the instance's data type. The following example illustrates this.

```
Dim arrayA( ) As Byte = New Byte(12) {}
Dim arrayB( ) As Byte = New Byte(100) {}
Dim arrayC( ) As Short = New Short(100) {}
Dim arrayD( , ) As Short
Dim arrayE( , ) As Short = New Short(4, 10) {}
```

In the preceding example, array variables `arrayA` and `arrayB` are considered to be of the same data type — **Byte()** — even though they are initialized to different lengths. Variables `arrayB` and `arrayC` are not of the same type because their element types are different. Variables `arrayC` and `arrayD` are not of the same type because their ranks are different. Variables `arrayD` and `arrayE` have the same type — **Short(,)** — because their ranks and element types are the same, even though `arrayD` is not yet initialized.

For more information on arrays, see Arrays in Visual Basic.

Class Types

There is no single data type comprising all classes. Although one class can inherit from another class, each is a separate data type. Multiple instances of the same class are of the same data type. If you assign one class instance variable to another, not only do they have the same data type, they point to the same class instance in memory.

For more information on classes, see Objects and Classes in Visual Basic.

See Also

Data Types in Visual Basic
Elementary Data Types (Visual Basic)
Generic Types in Visual Basic (Visual Basic)
Value Types and Reference Types
Type Conversions in Visual Basic
Structures (Visual Basic)
Troubleshooting Data Types (Visual Basic)
How to: Hold More Than One Value in a Variable (Visual Basic)

How to: Hold More Than One Value in a Variable (Visual Basic)

Visual Studio 2015

A variable holds more than one value if you declare it to be of a *composite data type*.

Composite Data Types (Visual Basic) include structures, arrays, and classes. A variable of a composite data type can hold a combination of elementary data types and other composite types. Structures and classes can hold code as well as data.

To hold more than one value in a variable

1. Determine what composite data type you want to use for your variable.
2. If the composite data type is not already defined, define it so that your variable can use it.
    - Define a structure with a Structure Statement.
    - Define an array with a Dim Statement (Visual Basic).
    - Define a class with a Class Statement (Visual Basic).
3. Declare your variable with a **Dim** statement.
4. Follow the variable name with an **As** clause.
5. Follow the **As** keyword with the name of the appropriate composite data type.

See Also

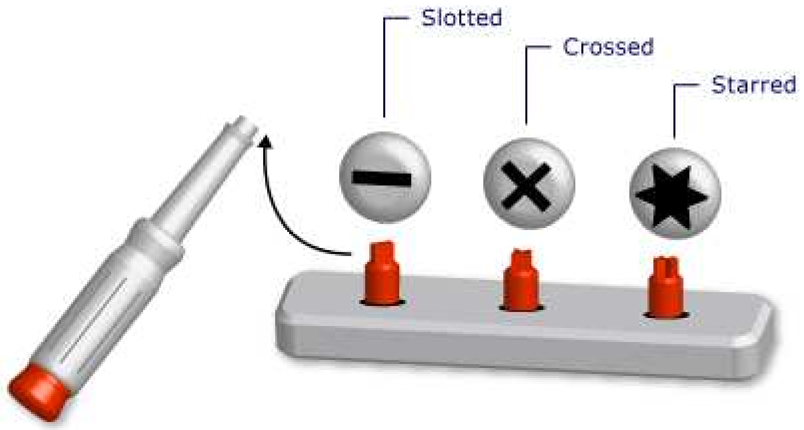
Data Type Summary (Visual Basic)
Type Characters (Visual Basic)
Composite Data Types (Visual Basic)
Structures (Visual Basic)
Arrays in Visual Basic
Objects and Classes in Visual Basic
Value Types and Reference Types

Generic Types in Visual Basic (Visual Basic)

Visual Studio 2015

A *generic type* is a single programming element that adapts to perform the same functionality for a variety of data types. When you define a generic class or procedure, you do not have to define a separate version for each data type for which you might want to perform that functionality.

An analogy is a screwdriver set with removable heads. You inspect the screw you need to turn and select the correct head for that screw (slotted, crossed, starred). Once you insert the correct head in the screwdriver handle, you perform the exact same function with the screwdriver, namely turning the screw.



Screwdriver set as a generic tool

When you define a generic type, you parameterize it with one or more data types. This allows the using code to tailor the data types to its requirements. Your code can declare several different programming elements from the generic element, each one acting on a different set of data types. But the declared elements all perform the identical logic, no matter what data types they are using.

For example, you might want to create and use a queue class that operates on a specific data type such as **String**. You can declare such a class from System.Collections.Generic.Queue(Of T), as the following example shows.

VB

```
Public stringQ As New System.Collections.Generic.Queue(Of String)
```

You can now use `stringQ` to work exclusively with **String** values. Because `stringQ` is specific for **String** instead of being generalized for **Object** values, you do not have late binding or type conversion. This saves execution time and reduces run-time errors.

For more information on using a generic type, see How to: Use a Generic Class (Visual Basic).

Example of a Generic Class

The following example shows a skeleton definition of a generic class.

VB

```
Public Class classHolder(Of t)
    Public Sub processNewItem(ByVal newItem As t)
        Dim tempItem As t
        ' Insert code that processes an item of data type t.
    End Sub
End Class
```

In the preceding skeleton, `t` is a *type parameter*, that is, a placeholder for a data type that you supply when you declare the class. Elsewhere in your code, you can declare various versions of `classHolder` by supplying various data types for `t`. The following example shows two such declarations.

VB

```
Public integerClass As New classHolder(Of Integer)
Friend stringClass As New classHolder(Of String)
```

The preceding statements declare *constructed classes*, in which a specific type replaces the type parameter. This replacement is propagated throughout the code within the constructed class. The following example shows what the `processNewItem` procedure looks like in `integerClass`.

VB

```
Public Sub processNewItem(ByVal newItem As Integer)
    Dim tempItem As Integer
    ' Inserted code now processes an Integer item.
End Sub
```

For a more complete example, see How to: Define a Class That Can Provide Identical Functionality on Different Data Types (Visual Basic).

Eligible Programming Elements

You can define and use generic classes, structures, interfaces, procedures, and delegates. Note that the .NET Framework defines several generic classes, structures, and interfaces that represent commonly used generic elements. The System.Collections.Generic namespace provides dictionaries, lists, queues, and stacks. Before defining your own generic element, see if it is already available in System.Collections.Generic.

Procedures are not types, but you can define and use generic procedures. See Generic Procedures in Visual Basic.

Advantages of Generic Types

A generic type serves as a basis for declaring several different programming elements, each of which operates on a specific data type. The alternatives to a generic type are:

1. A single type operating on the **Object** data type.

2. A set of *type-specific* versions of the type, each version individually coded and operating on one specific data type such as **String**, **Integer**, or a user-defined type such as `customer`.

A generic type has the following advantages over these alternatives:

- **Type Safety.** Generic types enforce compile-time type checking. Types based on **Object** accept any data type, and you must write code to check whether an input data type is acceptable. With generic types, the compiler can catch type mismatches before run time.

- **Performance.** Generic types do not have to *box* and *unbox* data, because each one is specialized for one data type. Operations based on **Object** must box input data types to convert them to **Object** and unbox data destined for output. Boxing and unboxing reduce performance.

  Types based on **Object** are also late-bound, which means that accessing their members requires extra code at run time. This also reduces performance.

- **Code Consolidation.** The code in a generic type has to be defined only once. A set of type-specific versions of a type must replicate the same code in each version, with the only difference being the specific data type for that version. With generic types, the type-specific versions are all generated from the original generic type.

- **Code Reuse.** Code that does not depend on a particular data type can be reused with various data types if it is generic. You can often reuse it even with a data type that you did not originally predict.

- **IDE Support.** When you use a constructed type declared from a generic type, the integrated development environment (IDE) can give you more support while you are developing your code. For example, IntelliSense can show you the type-specific options for an argument to a constructor or method.

- **Generic Algorithms.** Abstract algorithms that are type-independent are good candidates for generic types. For example, a generic procedure that sorts items using the IComparable interface can be used with any data type that implements IComparable.

Constraints

Although the code in a generic type definition should be as type-independent as possible, you might need to require a certain capability of any data type supplied to your generic type. For example, if you want to compare two items for the purpose of sorting or collating, their data type must implement the IComparable interface. You can enforce this requirement by adding a *constraint* to the type parameter.

Example of a Constraint

The following example shows a skeleton definition of a class with a constraint that requires the type argument to implement IComparable.

VB

```
Public Class itemManager(Of t As IComparable)
    ' Insert code that defines class members.
```

```
End Class
```

If subsequent code attempts to construct a class from `itemManager` supplying a type that does not implement IComparable, the compiler signals an error.

Types of Constraints

Your constraint can specify the following requirements in any combination:

- The type argument must implement one or more interfaces
- The type argument must be of the type of, or inherit from, at most one class
- The type argument must expose a parameterless constructor accessible to the code that creates objects from it
- The type argument must be a *reference type*, or it must be a *value type*

If you need to impose more than one requirement, you use a comma-separated *constraint list* inside braces (**{ }**). To require an accessible constructor, you include the New Operator (Visual Basic) keyword in the list. To require a reference type, you include the **Class** keyword; to require a value type, you include the **Structure** keyword.

For more information on constraints, see Type List (Visual Basic).

Example of Multiple Constraints

The following example shows a skeleton definition of a generic class with a constraint list on the type parameter. In the code that creates an instance of this class, the type argument must implement both the IComparable and IDisposable interfaces, be a reference type, and expose an accessible parameterless constructor.

VB

```
Public Class thisClass(Of t As {IComparable, IDisposable, Class, New})
    ' Insert code that defines class members.
End Class
```

Important Terms

Generic types introduce and use the following terms:

- *Generic Type*. A definition of a class, structure, interface, procedure, or delegate for which you supply at least one data type when you declare it.
- *Type Parameter*. In a generic type definition, a placeholder for a data type you supply when you declare the type.
- *Type Argument*. A specific data type that replaces a type parameter when you declare a constructed type from a

generic type.

- *Constraint*. A condition on a type parameter that restricts the type argument you can supply for it. A constraint can require that the type argument must implement a particular interface, be or inherit from a particular class, have an accessible parameterless constructor, or be a reference type or a value type. You can combine these constraints, but you can specify at most one class.

- *Constructed Type*. A class, structure, interface, procedure, or delegate declared from a generic type by supplying type arguments for its type parameters.

See Also

Data Types in Visual Basic
Type Characters (Visual Basic)
Value Types and Reference Types
Type Conversions in Visual Basic
Troubleshooting Data Types (Visual Basic)
Data Type Summary (Visual Basic)
Of Clause (Visual Basic)
As
Object Data Type
Covariance and Contravariance (C# and Visual Basic)
Iterators (C# and Visual Basic)

How to: Define a Class That Can Provide Identical Functionality on Different Data Types (Visual Basic)

Visual Studio 2015

You can define a class from which you can create objects that provide identical functionality on different data types. To do this, you specify one or more *type parameters* in the definition. The class can then serve as a template for objects that use various data types. A class defined in this way is called a *generic class*.

The advantage of defining a generic class is that you define it just once, and your code can use it to create many objects that use a wide variety of data types. This results in better performance than defining the class with the **Object** type.

In addition to classes, you can also define and use generic structures, interfaces, procedures, and delegates.

To define a class with a type parameter

1. Define the class in the normal way.
2. Add **(Of** *typeparameter***)** immediately after the class name to specify a type parameter.
3. If you have more than one type parameter, make a comma-separated list inside the parentheses. Do not repeat the **Of** keyword.
4. If your code performs operations on a type parameter other than simple assignment, follow that type parameter with an **As** clause to add one or more *constraints*. A constraint guarantees that the type supplied for that type parameter satisfies a requirement such as the following:
   - Supports an operation, such as **>**, that your code performs
   - Supports a member, such as a method, that your code accesses
   - Exposes a parameterless constructor

   If you do not specify any constraints, the only operations and members your code can use are those supported by the Object Data Type. For more information, see Type List (Visual Basic).
5. Identify every class member that is to be declared with a supplied type, and declare it **As** *typeparameter*. This applies to internal storage, procedure parameters, and return values.
6. Be sure your code uses only operations and methods that are supported by any data type it can supply to `itemType`.

   The following example defines a class that manages a very simple list. It holds the list in the internal array `items`, and the using code can declare the data type of the list elements. A parameterized constructor allows the using code to set the upper bound of `items`, and the default constructor sets this to 9 (for a total of 10 items).

   VB

```
Public Class simpleList(Of itemType)
  Private items() As itemType
  Private top As Integer
  Private nextp As Integer
  Public Sub New()
    Me.New(9)
  End Sub
  Public Sub New(ByVal t As Integer)
    MyBase.New()
    items = New itemType(t) {}
    top = t
    nextp = 0
  End Sub
  Public Sub add(ByVal i As itemType)
    insert(i, nextp)
  End Sub
  Public Sub insert(ByVal i As itemType, ByVal p As Integer)
    If p > nextp OrElse p < 0 Then
      Throw New System.ArgumentOutOfRangeException("p",
        " less than 0 or beyond next available list position")
    ElseIf nextp > top Then
      Throw New System.ArgumentException("No room to insert at ",
        "p")
    ElseIf p < nextp Then
      For j As Integer = nextp To p + 1 Step -1
        items(j) = items(j - 1)
      Next j
    End If
    items(p) = i
    nextp += 1
  End Sub
  Public Sub remove(ByVal p As Integer)
    If p >= nextp OrElse p < 0 Then
        Throw New System.ArgumentOutOfRangeException("p",
            " less than 0 or beyond last list item")
    ElseIf nextp = 0 Then
        Throw New System.ArgumentException("List empty; cannot remove ",
            "p")
    ElseIf p < nextp - 1 Then
        For j As Integer = p To nextp - 2
            items(j) = items(j + 1)
        Next j
    End If
    nextp -= 1
  End Sub
  Public ReadOnly Property listLength() As Integer
    Get
      Return nextp
    End Get
  End Property
  Public ReadOnly Property listItem(ByVal p As Integer) As itemType
    Get
      If p >= nextp OrElse p < 0 Then
```

```
        Throw New System.ArgumentOutOfRangeException("p",
          " less than 0 or beyond last list item")
        End If
      Return items(p)
    End Get
  End Property
End Class
```

You can declare a class from `simpleList` to hold a list of **Integer** values, another class to hold a list of **String** values, and another to hold **Date** values. Except for the data type of the list members, objects created from all these classes behave identically.

The type argument that the using code supplies to `itemType` can be an intrinsic type such as **Boolean** or **Double**, a structure, an enumeration, or any type of class, including one that your application defines.

You can test the class `simpleList` with the following code.

VB

```
Public Sub useSimpleList()
  Dim iList As New simpleList(Of Integer)(2)
  Dim sList As New simpleList(Of String)(3)
  Dim dList As New simpleList(Of Date)(2)
  iList.add(10)
  iList.add(20)
  iList.add(30)
  sList.add("First")
  sList.add("extra")
  sList.add("Second")
  sList.add("Third")
  sList.remove(1)
  dList.add(#1/1/2003#)
  dList.add(#3/3/2003#)
  dList.insert(#2/2/2003#, 1)
  Dim s =
    "Simple list of 3 Integer items (reported length " &
     CStr(iList.listLength) & "):" &
     vbCrLf & CStr(iList.listItem(0)) &
     vbCrLf & CStr(iList.listItem(1)) &
     vbCrLf & CStr(iList.listItem(2)) &
     vbCrLf &
     "Simple list of 4 - 1 String items (reported length " &
     CStr(sList.listLength) & "):" &
     vbCrLf & CStr(sList.listItem(0)) &
     vbCrLf & CStr(sList.listItem(1)) &
     vbCrLf & CStr(sList.listItem(2)) &
     vbCrLf &
     "Simple list of 2 + 1 Date items (reported length " &
     CStr(dList.listLength) & "):" &
     vbCrLf & CStr(dList.listItem(0)) &
     vbCrLf & CStr(dList.listItem(1)) &
     vbCrLf & CStr(dList.listItem(2))
  MsgBox(s)
```

```
    End Sub
```

See Also

Data Types in Visual Basic
Generic Types in Visual Basic (Visual Basic)
Language Independence and Language-Independent Components
Of Clause (Visual Basic)
Type List (Visual Basic)
How to: Use a Generic Class (Visual Basic)
Object Data Type

How to: Use a Generic Class (Visual Basic)

Visual Studio 2015

A class that takes *type parameters* is called a *generic class*. If you are using a generic class, you can generate a *constructed class* from it by supplying a *type argument* for each of these parameters. You can then declare a variable of the constructed class type, and you can create an instance of the constructed class and assign it to that variable.

In addition to classes, you can also define and use generic structures, interfaces, procedures, and delegates.

The following procedure takes a generic class defined in the .NET Framework and creates an instance from it.

To use a class that takes a type parameter

1. At the beginning of your source file, include an Imports Statement (.NET Namespace and Type) to import the System.Collections.Generic namespace. This allows you to refer to the System.Collections.Generic.Queue(Of T) class without having to fully qualify it to differentiate it from other queue classes such as System.Collections.Queue.

2. Create the object in the normal way, but add **(Of** *type***)** immediately after the class name.

   The following example uses the same class (System.Collections.Generic.Queue(Of T)) to create two queue objects that hold items of different data types. It adds items to the end of each queue and then removes and displays items from the front of each queue.

   VB

```
Public Sub usequeue()
  Dim queueDouble As New System.Collections.Generic.Queue(Of Double)
  Dim queueString As New System.Collections.Generic.Queue(Of String)
  queueDouble.Enqueue(1.1)
  queueDouble.Enqueue(2.2)
  queueDouble.Enqueue(3.3)
  queueDouble.Enqueue(4.4)
  queueString.Enqueue("First string of three")
  queueString.Enqueue("Second string of three")
  queueString.Enqueue("Third string of three")
  Dim s As String = "Queue of Double items (reported length " &
      CStr(queueDouble.Count) & "):"
  For i As Integer = 1 To queueDouble.Count
    s &= vbCrLf & CStr(queueDouble.Dequeue())
  Next i
  s &= vbCrLf & "Queue of String items (reported length " &
      CStr(queueString.Count) & "):"
  For i As Integer = 1 To queueString.Count
    s &= vbCrLf & queueString.Dequeue()
  Next i
  MsgBox(s)
End Sub
```

See Also

Data Types in Visual Basic
Generic Types in Visual Basic (Visual Basic)
Language Independence and Language-Independent Components
Of Clause (Visual Basic)
Imports Statement (.NET Namespace and Type)
How to: Define a Class That Can Provide Identical Functionality on Different Data Types (Visual Basic)
Iterators (C# and Visual Basic)

Generic Procedures in Visual Basic

Visual Studio 2015

A *generic procedure*, also called a *generic method*, is a procedure defined with at least one type parameter. This allows the calling code to tailor the data types to its requirements each time it calls the procedure.

A procedure is not generic simply by virtue of being defined inside a generic class or a generic structure. To be generic, the procedure must take at least one type parameter, in addition to any normal parameters it might take. A generic class or structure can contain nongeneric procedures, and a nongeneric class, structure, or module can contain generic procedures.

A generic procedure can use its type parameters in its normal parameter list, in its return type if it has one, and in its procedure code.

Type Inference

You can call a generic procedure without supplying any type arguments at all. If you call it this way, the compiler attempts to determine the appropriate data types to pass to the procedure's type arguments. This is called *type inference*. The following code shows a call in which the compiler infers that it should pass type **String** to the type parameter `t`.

VB

```
Public Sub testSub(Of t)(ByVal arg As t)
End Sub
Public Sub callTestSub()
    testSub("Use this string")
End Sub
```

If the compiler cannot infer the type arguments from the context of your call, it reports an error. One possible cause of such an error is an array rank mismatch. For example, suppose you define a normal parameter as an array of a type parameter. If you call the generic procedure supplying an array of a different rank (number of dimensions), the mismatch causes type inference to fail. The following code shows a call in which a two-dimensional array is passed to a procedure that expects a one-dimensional array.

```
Public Sub demoSub(Of t)(ByVal arg() As t)

End Sub

Public Sub callDemoSub()

Dim twoDimensions(,) As Integer

demoSub(twoDimensions)

End Sub
```

You can invoke type inference only by omitting all the type arguments. If you supply one type argument, you must supply them all.

Type inference is supported only for generic procedures. You cannot invoke type inference on generic classes, structures, interfaces, or delegates.

Example

Description

The following example defines a generic **Function** procedure to find a particular element in an array. It defines one type parameter and uses it to construct the two parameters in the parameter list.

Code

VB

```
Public Function findElement(Of T As IComparable) (
        ByVal searchArray As T(), ByVal searchValue As T) As Integer

    If searchArray.GetLength(0) > 0 Then
        For i As Integer = 0 To searchArray.GetUpperBound(0)
            If searchArray(i).CompareTo(searchValue) = 0 Then Return i
        Next i
    End If

    Return -1
End Function
```

Comments

The preceding example requires the ability to compare `searchValue` against each element of `searchArray`. To guarantee this ability, it constrains the type parameter `T` to implement the IComparable(Of T) interface. The code uses the CompareTo method instead of the = operator, because there is no guarantee that a type argument supplied for `T` supports the = operator.

You can test the `findElement` procedure with the following code.

VB

```
Public Sub tryFindElement()
    Dim stringArray() As String = {"abc", "def", "xyz"}
    Dim stringSearch As String = "abc"
    Dim integerArray() As Integer = {7, 8, 9}
    Dim integerSearch As Integer = 8
    Dim dateArray() As Date = {#4/17/1969#, #9/20/1998#, #5/31/2004#}
    Dim dateSearch As Date = Microsoft.VisualBasic.DateAndTime.Today
    MsgBox(CStr(findElement(Of String)(stringArray, stringSearch)))
    MsgBox(CStr(findElement(Of Integer)(integerArray, integerSearch)))
```

```
        MsgBox(CStr(findElement(Of Date)(dateArray, dateSearch)))
    End Sub
```

The preceding calls to **MsgBox** display "0", "1", and "-1" respectively.

See Also

Generic Types in Visual Basic (Visual Basic)
How to: Define a Class That Can Provide Identical Functionality on Different Data Types (Visual Basic)
How to: Use a Generic Class (Visual Basic)
Procedures in Visual Basic
Procedure Parameters and Arguments (Visual Basic)
Type List (Visual Basic)
Parameter List (Visual Basic)

Nullable Value Types (Visual Basic)

Visual Studio 2015

Sometimes you work with a value type that does not have a defined value in certain circumstances. For example, a field in a database might have to distinguish between having an assigned value that is meaningful and not having an assigned value. Value types can be extended to take either their normal values or a null value. Such an extension is called a *nullable type*.

Each nullable type is constructed from the generic Nullable(Of T) structure. Consider a database that tracks work-related activities. The following example constructs a nullable **Boolean** type and declares a variable of that type. You can write the declaration in three ways:

VB

```
Dim ridesBusToWork1? As Boolean
Dim ridesBusToWork2 As Boolean?
Dim ridesBusToWork3 As Nullable(Of Boolean)
```

The variable `ridesBusToWork` can hold a value of **True**, a value of **False**, or no value at all. Its initial default value is no value at all, which in this case could mean that the information has not yet been obtained for this person. In contrast, **False** could mean that the information has been obtained and the person does not ride the bus to work.

You can declare variables and properties with nullable types, and you can declare an array with elements of a nullable type. You can declare procedures with nullable types as parameters, and you can return a nullable type from a **Function** procedure.

You cannot construct a nullable type on a reference type such as an array, a **String**, or a class. The underlying type must be a value type. For more information, see Value Types and Reference Types.

Using a Nullable Type Variable

The most important members of a nullable type are its HasValue and Value properties. For a variable of a nullable type, HasValue tells you whether the variable contains a defined value. If HasValue is **True**, you can read the value from Value. Note that both HasValue and Value are **ReadOnly** properties.

Default Values

When you declare a variable with a nullable type, its HasValue property has a default value of **False**. This means that by default the variable has no defined value, instead of the default value of its underlying value type. In the following example, the variable `numberOfChildren` initially has no defined value, even though the default value of the **Integer** type is 0.

VB

```
Dim numberOfChildren? As Integer
```

A null value is useful to indicate an undefined or unknown value. If `numberOfChildren` had been declared as **Integer**, there would be no value that could indicate that the information is not currently available.

Storing Values

You store a value in a variable or property of a nullable type in the typical way. The following example assigns a value to the variable `numberOfChildren` declared in the previous example.

VB

```
numberOfChildren = 2
```

If a variable or property of a nullable type contains a defined value, you can cause it to revert to its initial state of not having a value assigned. You do this by setting the variable or property to **Nothing**, as the following example shows.

VB

```
numberOfChildren = Nothing
```

Note

Although you can assign **Nothing** to a variable of a nullable type, you cannot test it for **Nothing** by using the equal sign. Comparison that uses the equal sign, `someVar = Nothing`, always evaluates to **Nothing**. You can test the variable's HasValue property for **False**, or test by using the **Is** or **IsNot** operator.

Retrieving Values

To retrieve the value of a variable of a nullable type, you should first test its HasValue property to confirm that it has a value. If you try to read the value when HasValue is **False**, Visual Basic throws an InvalidOperationException exception. The following example shows the recommended way to read the variable `numberOfChildren` of the previous examples.

VB

```
If numberOfChildren.HasValue Then
    MsgBox("There are " & CStr(numberOfChildren) & " children.")
Else
    MsgBox("It is not known how many children there are.")
End If
```

Comparing Nullable Types

When nullable **Boolean** variables are used in Boolean expressions, the result can be **True**, **False**, or **Nothing**. The following is the truth table for **And** and **Or**. Because b1 and b2 now have three possible values, there are nine combinations to evaluate.

b1	b2	b1 And b2	b1 Or b2
Nothing	Nothing	Nothing	Nothing
Nothing	True	Nothing	True
Nothing	False	False	Nothing
True	Nothing	Nothing	True
True	True	True	True
True	False	False	True
False	Nothing	False	Nothing
False	True	False	True
False	False	False	False

When the value of a Boolean variable or expression is **Nothing**, it is neither **true** nor **false**. Consider the following example.

VB

```
Dim b1? As Boolean
Dim b2? As Boolean
b1 = True
b2 = Nothing

' The following If statement displays "Expression is not true".
If (b1 And b2) Then
    Console.WriteLine("Expression is true")
Else
    Console.WriteLine("Expression is not true")
End If

' The following If statement displays "Expression is not false".
If Not (b1 And b2) Then
    Console.WriteLine("Expression is false")
Else
    Console.WriteLine("Expression is not false")
End If
```

In this example, b1 And b2 evaluates to **Nothing**. As a result, the Else clause is executed in each **If** statement, and the output is as follows:

```
Expression is not true

Expression is not false
```

Note

AndAlso and **OrElse**, which use short-circuit evaluation, must evaluate their second operands when the first evaluates to **Nothing**.

Propagation

If one or both of the operands of an arithmetic, comparison, shift, or type operation is nullable, the result of the operation is also nullable. If both operands have values that are not **Nothing**, the operation is performed on the underlying values of the operands, as if neither were a nullable type. In the following example, variables `compare1` and `sum1` are implicitly typed. If you rest the mouse pointer over them, you will see that the compiler infers nullable types for both of them.

VB

```
' Variable n is a nullable type, but both m and n have proper values.
Dim m As Integer = 3
Dim n? As Integer = 2

' The comparison evaluated is 3>2, but compare1 is inferred to be of
' type Boolean?.
Dim compare1 = m > n
' The values summed are 3 and 2, but sum1 is inferred to be of type Integer?.
Dim sum1 = m + n

' The following line displays: 3 * 2 * 5 * True
Console.WriteLine(m & " * " & n & " * " & sum1 & " * " & compare1)
```

If one or both operands have a value of **Nothing**, the result will be **Nothing**.

VB

```
' Change the value of n to Nothing.
n = Nothing

Dim compare2 = m > n
Dim sum2 = m + n

' Because the values of n, compare2, and sum2 are all Nothing, the
' following line displays 3 * * *
Console.WriteLine(m & " * " & n & " * " & sum2 & " * " & compare2)
```

Using Nullable Types with Data

A database is one of the most important places to use nullable types. Not all database objects currently support nullable types, but the designer-generated table adapters do. See "TableAdapter Support for Nullable Types" in TableAdapter Overview.

See Also

InvalidOperationException
HasValue
Using Nullable Types (C# Programming Guide)
Data Types in Visual Basic
Value Types and Reference Types
Troubleshooting Data Types (Visual Basic)
TableAdapter Overview
If Operator (Visual Basic)
Local Type Inference (Visual Basic)
Is Operator (Visual Basic)
IsNot Operator (Visual Basic)

Value Types and Reference Types

Visual Studio 2015

In Visual Basic, data types are implemented based on their classification. The Visual Basic data types can be classified according to whether a variable of a particular type stores its own data or a pointer to the data. If it stores its own data it is a *value type*; if it holds a pointer to data elsewhere in memory it is a *reference type*.

Value Types

A data type is a *value type* if it holds the data within its own memory allocation. Value types include the following:

- All numeric data types
- **Boolean**, **Char**, and **Date**
- All structures, even if their members are reference types
- Enumerations, since their underlying type is always **SByte**, **Short**, **Integer**, **Long**, **Byte**, **UShort**, **UInteger**, or **ULong**

Every structure is a value type, even if it contains reference type members. For this reason, value types such as **Char** and **Integer** are implemented by .NET Framework structures.

You can declare a value type by using the reserved keyword, for example, **Decimal**. You can also use the **New** keyword to initialize a value type. This is especially useful if the type has a constructor that takes parameters. An example of this is the Decimal(Int32, Int32, Int32, Boolean, Byte) constructor, which builds a new **Decimal** value from the supplied parts.

Reference Types

A *reference type* contains a pointer to another memory location that holds the data. Reference types include the following:

- **String**
- All arrays, even if their elements are value types
- Class types, such as Form
- Delegates

A class is a *reference type*. For this reason, reference types such as **Object** and **String** are supported by .NET Framework classes. Note that every array is a reference type, even if its members are value types.

Since every reference type represents an underlying .NET Framework class, you must use the New Operator (Visual Basic) keyword when you initialize it. The following statement initializes an array.

```
Dim totals() As Single = New Single(8) {}
```

Elements That Are Not Types

The following programming elements do not qualify as types, because you cannot specify any of them as a data type for a declared element:

- Namespaces
- Modules
- Events
- Properties and procedures
- Variables, constants, and fields

Working with the Object Data Type

You can assign either a reference type or a value type to a variable of the **Object** data type. An **Object** variable always holds a pointer to the data, never the data itself. However, if you assign a value type to an **Object** variable, it behaves as if it holds its own data. For more information, see Object Data Type.

You can find out whether an **Object** variable is acting as a reference type or a value type by passing it to the IsReference method in the Information class of the Microsoft.VisualBasic namespace. Information.IsReference returns **True** if the content of the **Object** variable represents a reference type.

See Also

Nullable Value Types (Visual Basic)
Type Conversions in Visual Basic
Structure Statement
Efficient Use of Data Types (Visual Basic)
Object Data Type
Data Types in Visual Basic

Type Conversions in Visual Basic

Visual Studio 2015

The process of changing a value from one data type to another type is called *conversion*. Conversions are either *widening* or *narrowing*, depending on the data capacities of the types involved. They are also *implicit* or *explicit*, depending on the syntax in the source code.

In This Section

Widening and Narrowing Conversions (Visual Basic)
Explains conversions classified by whether the destination type can hold the data.

Implicit and Explicit Conversions (Visual Basic)
Discusses conversions classified by whether Visual Basic performs them automatically.

Conversions Between Strings and Other Types (Visual Basic)
Illustrates converting between strings and numeric, **Boolean**, or date/time values.

How to: Convert an Object to Another Type in Visual Basic
Shows how to convert an **Object** variable to any other data type.

Array Conversions (Visual Basic)
Steps you through the process of converting between arrays of different data types.

Related Sections

Data Types in Visual Basic
Introduces the Visual Basic data types and describes how to use them.

Data Type Summary (Visual Basic)
Lists the elementary data types supplied by Visual Basic.

Troubleshooting Data Types (Visual Basic)
Discusses some common problems that can arise when working with data types.

Widening and Narrowing Conversions (Visual Basic)

Visual Studio 2015

An important consideration with a type conversion is whether the result of the conversion is within the range of the destination data type.

A *widening conversion* changes a value to a data type that can allow for any possible value of the original data. Widening conversions preserve the source value but can change its representation. This occurs if you convert from an integral type to **Decimal**, or from **Char** to **String**.

A *narrowing conversion* changes a value to a data type that might not be able to hold some of the possible values. For example, a fractional value is rounded when it is converted to an integral type, and a numeric type being converted to **Boolean** is reduced to either **True** or **False**.

Widening Conversions

The following table shows the standard widening conversions.

Data type	Widens to data types [1]
SByte	**SByte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**
Byte	**Byte**, **Short**, **UShort**, **Integer**, **UInteger**, **Long**, **ULong**, **Decimal**, **Single**, **Double**
Short	**Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**
UShort	**UShort**, **Integer**, **UInteger**, **Long**, **ULong**, **Decimal**, **Single**, **Double**
Integer	**Integer**, **Long**, **Decimal**, **Single**, **Double**[2]
UInteger	**UInteger**, **Long**, **ULong**, **Decimal**, **Single**, **Double** [2]
Long	**Long**, **Decimal**, **Single**, **Double** [2]
ULong	**ULong**, **Decimal**, **Single**, **Double** [2]
Decimal	**Decimal**, **Single**, **Double** [2]
Single	**Single**, **Double**

Double	**Double**
Any enumerated type (Enum)	Its underlying integral type and any type to which the underlying type widens.
Char	**Char**, **String**
Char array	**Char** array, **String**
Any type	Object
Any derived type	Any base type from which it is derived [3].
Any type	Any interface it implements.
Nothing	Any data type or object type.

[1] By definition, every data type widens to itself.

[2] Conversions from **Integer**, **UInteger**, **Long**, **ULong**, or **Decimal** to **Single** or **Double** might result in loss of precision, but never in loss of magnitude. In this sense they do not incur information loss.

[3] It might seem surprising that a conversion from a derived type to one of its base types is widening. The justification is that the derived type contains all the members of the base type, so it qualifies as an instance of the base type. In the opposite direction, the base type does not contain any new members defined by the derived type.

Widening conversions always succeed at run time and never incur data loss. You can always perform them implicitly, whether the Option Strict Statement sets the type checking switch to **On** or to **Off**.

Narrowing Conversions

The standard narrowing conversions include the following:

- The reverse directions of the widening conversions in the preceding table (except that every type widens to itself)
- Conversions in either direction between Boolean and any numeric type
- Conversions from any numeric type to any enumerated type (**Enum**)
- Conversions in either direction between String and any numeric type, **Boolean**, or Date
- Conversions from a data type or object type to a type derived from it

Narrowing conversions do not always succeed at run time, and can fail or incur data loss. An error occurs if the destination data type cannot receive the value being converted. For example, a numeric conversion can result in an overflow. The compiler does not allow you to perform narrowing conversions implicitly unless the Option Strict Statement sets the type checking switch to **Off**.

Note

The narrowing-conversion error is suppressed for conversions from the elements in a **For Each…Next** collection to the loop control variable. For more information and examples, see the "Narrowing Conversions" section in For Each...Next Statement (Visual Basic).

When to Use Narrowing Conversions

You use a narrowing conversion when you know the source value can be converted to the destination data type without error or data loss. For example, if you have a **String** that you know contains either "True" or "False," you can use the **CBool** keyword to convert it to **Boolean**.

Exceptions During Conversion

Because widening conversions always succeed, they do not throw exceptions. Narrowing conversions, when they fail, most commonly throw the following exceptions:

- InvalidCastException — if no conversion is defined between the two types
- OverflowException — (integral types only) if the converted value is too large for the target type

If a class or structure defines a CType Function (Visual Basic) to serve as a conversion operator to or from that class or structure, that **CType** can throw any exception it deems appropriate. In addition, that **CType** might call Visual Basic functions or .NET Framework methods, which in turn could throw a variety of exceptions.

Changes During Reference Type Conversions

A conversion from a *reference type* copies only the pointer to the value. The value itself is neither copied nor changed in any way. The only thing that can change is the data type of the variable holding the pointer. In the following example, the data type is converted from the derived class to its base class, but the object that both variables now point to is unchanged.

```
' Assume class cSquare inherits from class cShape.
Dim shape As cShape
Dim square As cSquare = New cSquare
' The following statement performs a widening
' conversion from a derived class to its base class.
shape = square
```

See Also

Data Types in Visual Basic
Type Conversions in Visual Basic
Implicit and Explicit Conversions (Visual Basic)
Conversions Between Strings and Other Types (Visual Basic)
How to: Convert an Object to Another Type in Visual Basic
Array Conversions (Visual Basic)
Data Type Summary (Visual Basic)
Type Conversion Functions (Visual Basic)

Implicit and Explicit Conversions (Visual Basic)

Visual Studio 2015

An *implicit conversion* does not require any special syntax in the source code. In the following example, Visual Basic implicitly converts the value of k to a single-precision floating-point value before assigning it to q.

```
Dim k As Integer
Dim q As Double
' Integer widens to Double, so you can do this with Option Strict On.
k = 432
q = k
```

An *explicit conversion* uses a type conversion keyword. Visual Basic provides several such keywords, which coerce an expression in parentheses to the desired data type. These keywords act like functions, but the compiler generates the code inline, so execution is slightly faster than with a function call.

In the following extension of the preceding example, the **CInt** keyword converts the value of q back to an integer before assigning it to k.

```
' q had been assigned the value 432 from k.
q = Math.Sqrt(q)
k = CInt(q)
' k now has the value 21 (rounded square root of 432).
```

Conversion Keywords

The following table shows the available conversion keywords.

Type conversion keyword	Converts an expression to data type	Allowable data types of expression to be converted
CBool	Boolean Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **String**, **Object**
CByte	Byte Data Type (Visual Basic)	Any numeric type (including **SByte** and enumerated types), **Boolean**, **String**, **Object**

CChar	Char Data Type (Visual Basic)	**String**, **Object**
CDate	Date Data Type (Visual Basic)	**String**, **Object**
CDbl	Double Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CDec	Decimal Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CInt	Integer Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CLng	Long Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CObj	Object Data Type	Any type
CSByte	SByte Data Type (Visual Basic)	Any numeric type (including **Byte** and enumerated types), **Boolean**, **String**, **Object**
CShort	Short Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CSng	Single Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CStr	String Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **Char**, **Char** array, **Date**, **Object**
CType	Type specified following the comma (**,**)	When converting to an *elementary data type* (including an array of an elementary type), the same types as allowed for the corresponding conversion keyword When converting to a *composite data type*, the interfaces it implements and the classes from which it inherits When converting to a class or structure on which you have overloaded **CType**, that class or structure
CUInt	UInteger Data Type	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CULng	ULong Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**
CUShort	UShort Data Type (Visual Basic)	Any numeric type (including **Byte**, **SByte**, and enumerated types), **Boolean**, **String**, **Object**

The CType Function

The CType Function (Visual Basic) operates on two arguments. The first is the expression to be converted, and the second is the destination data type or object class. Note that the first argument must be an expression, not a type.

CType is an *inline function*, meaning the compiled code makes the conversion, often without generating a function call. This improves performance.

For a comparison of **CType** with the other type conversion keywords, see DirectCast Operator (Visual Basic) and TryCast Operator (Visual Basic).

Elementary Types

The following example demonstrates the use of **CType**.

```
k = CType(q, Integer)
' The following statement coerces w to the specific object class Label.
f = CType(w, Label)
```

Composite Types

You can use **CType** to convert values to composite data types as well as to elementary types. You can also use it to coerce an object class to the type of one of its interfaces, as in the following example.

```
' Assume class cZone implements interface iZone.
Dim h As Object
' The first argument to CType must be an expression, not a type.
Dim cZ As cZone
' The following statement coerces a cZone object to its interface iZone.
h = CType(cZ, iZone)
```

Array Types

CType can also convert array data types, as in the following example.

```
Dim v() As classV
Dim obArray() As Object
' Assume some object array has been assigned to obArray.
' Check for run-time type compatibility.
If TypeOf obArray Is classV()
    ' obArray can be converted to classV.
    v = CType(obArray, classV())
    End If
```

For more information and an example, see Array Conversions (Visual Basic).

Types Defining CType

You can define **CType** on a class or structure you have defined. This allows you to convert values to and from the type of your class or structure. For more information and an example, see How to: Define a Conversion Operator (Visual Basic).

Note

Values used with a conversion keyword must be valid for the destination data type, or an error occurs. For example, if you attempt to convert a **Long** to an **Integer**, the value of the **Long** must be within the valid range for the **Integer** data type.

Caution

Specifying **CType** to convert from one class type to another fails at run time if the source type does not derive from the destination type. Such a failure throws an InvalidCastException exception.

However, if one of the types is a structure or class you have defined, and if you have defined **CType** on that structure or class, a conversion can succeed if it satisfies the requirements of your **CType**. See How to: Define a Conversion Operator (Visual Basic).

Performing an explicit conversion is also known as *casting* an expression to a given data type or object class.

See Also

Type Conversions in Visual Basic
Conversions Between Strings and Other Types (Visual Basic)
How to: Convert an Object to Another Type in Visual Basic
Structures (Visual Basic)
Data Type Summary (Visual Basic)
Type Conversion Functions (Visual Basic)
Troubleshooting Data Types (Visual Basic)

Conversions Between Strings and Other Types (Visual Basic)

Visual Studio 2015

You can convert a numeric, **Boolean**, or date/time value to a **String**. You can also convert in the reverse direction — from a string value to numeric, **Boolean**, or **Date** — provided the contents of the string can be interpreted as a valid value of the destination data type. If they cannot, a run-time error occurs.

The conversions for all these assignments, in either direction, are narrowing conversions. You should use the type conversion keywords (**CBool**, **CByte**, **CDate**, **CDbl**, **CDec**, **CInt**, **CLng**, **CSByte**, **CShort**, **CSng**, **CStr**, **CUInt**, **CULng**, **CUShort**, and **CType**). The Format and Val functions give you additional control over conversions between strings and numbers.

If you have defined a class or structure, you can define type conversion operators between **String** and the type of your class or structure. For more information, see How to: Define a Conversion Operator (Visual Basic).

Conversion of Numbers to Strings

You can use the **Format** function to convert a number to a formatted string, which can include not only the appropriate digits but also formatting symbols such as a currency sign (such as **$**), thousands separators or *digit grouping symbols* (such as **,**), and a decimal separator (such as **.**). **Format** automatically uses the appropriate symbols according to the **Regional Options** settings specified in the Windows **Control Panel**.

Note that the concatenation (**&**) operator can convert a number to a string implicitly, as the following example shows.

```
' The following statement converts count to a String value.
Str = "The total count is " & count
```

Conversion of Strings to Numbers

You can use the **Val** function to explicitly convert the digits in a string to a number. **Val** reads the string until it encounters a character other than a digit, space, tab, line feed, or period. The sequences "&O" and "&H" alter the base of the number system and terminate the scanning. Until it stops reading, **Val** converts all appropriate characters to a numeric value. For example, the following statement returns the value `141.825`.

```
Val(" 14 1.825 miles")
```

When Visual Basic converts a string to a numeric value, it uses the **Regional Options** settings specified in the Windows **Control Panel** to interpret the thousands separator, decimal separator, and currency symbol. This means that a conversion might succeed under one setting but not another. For example, `"$14.20"` is acceptable in the English (United States) locale but not in any French locale.

See Also

Type Conversions in Visual Basic
Widening and Narrowing Conversions (Visual Basic)
Implicit and Explicit Conversions (Visual Basic)
How to: Convert an Object to Another Type in Visual Basic
Array Conversions (Visual Basic)
Data Type Summary (Visual Basic)
Type Conversion Functions (Visual Basic)
Introduction to International Applications Based on the .NET Framework

How to: Convert an Object to Another Type in Visual Basic

Visual Studio 2015

You convert an **Object** variable to another data type by using a conversion keyword such as CType Function (Visual Basic).

Example

The following example converts an **Object** variable to an **Integer** and a **String**.

```
Public Sub objectConversion(ByVal anObject As Object)
    Dim anInteger As Integer
    Dim aString As String
    anInteger = CType(anObject, Integer)
    aString = CType(anObject, String)
End Sub
```

If you know that the contents of an **Object** variable are of a particular data type, it is better to convert the variable to that data type. If you continue to use the **Object** variable, you incur either *boxing* and *unboxing* (for a value type) or *late binding* (for a reference type). These operations all take extra execution time and make your performance slower.

Compiling the Code

This example requires:

- A reference to the System namespace.

See Also

Object
Type Conversions in Visual Basic
Widening and Narrowing Conversions (Visual Basic)
Implicit and Explicit Conversions (Visual Basic)
Conversions Between Strings and Other Types (Visual Basic)
Array Conversions (Visual Basic)
Structures (Visual Basic)
Data Type Summary (Visual Basic)
Type Conversion Functions (Visual Basic)

Array Conversions (Visual Basic)

Visual Studio 2015

You can convert an array type to a different array type provided you meet the following conditions:

- **Equal Rank.** The ranks of the two arrays must be the same, that is, they must have the same number of dimensions. However, the lengths of the respective dimensions do not need to be the same.
- **Element Data Type.** The data types of the elements of both arrays must be reference types. You cannot convert an **Integer** array to a **Long** array, or even to an **Object** array, because at least one value type is involved. For more information, see Value Types and Reference Types.
- **Convertibility.** A conversion, either widening or narrowing, must be possible between the element types of the two arrays. An example that fails this requirement is an attempted conversion between a **String** array and an array of a class derived from System.Attribute. These two types have nothing in common, and no conversion of any kind exists between them.

A conversion of one array type to another is widening or narrowing depending on whether the conversion of the respective elements is widening or narrowing. For more information, see Widening and Narrowing Conversions (Visual Basic).

Conversion to an Object Array

When you declare an **Object** array without initializing it, its element type is **Object** as long as it remains uninitialized. When you set it to an array of a specific class, it takes on the type of that class. However, its underlying type is still **Object**, and you can subsequently set it to another array of an unrelated class. Since all classes derive from **Object**, you can change the array's element type from any class to any other class.

In the following example, no conversion exists between types `student` and **String**, but both derive from **Object**, so all assignments are valid.

```
' Assume student has already been defined as a class.
Dim testArray() As Object
' testArray is still an Object array at this point.
Dim names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}
testArray = New student(3) {}
' testArray is now of type student().
testArray = names
' testArray is now a String array.
```

Underlying Type of an Array

If you originally declare an array with a specific class, its underlying element type is that class. If you subsequently set it to an array of another class, there must be a conversion between the two classes.

In the following example, `students` is a `student` array. Since no conversion exists between **String** and `student`, the last statement fails.

```
Dim students() As student
Dim names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}
students = New Student(3) {}
' The following statement fails at compile time.
students = names
```

See Also

- Data Types in Visual Basic
- Type Conversions in Visual Basic
- Implicit and Explicit Conversions (Visual Basic)
- Conversions Between Strings and Other Types (Visual Basic)
- How to: Convert an Object to Another Type in Visual Basic
- Data Type Summary (Visual Basic)
- Type Conversion Functions (Visual Basic)
- Arrays in Visual Basic