Chapter 4 –Language Building Blocks

# Objectives

* Data Type Ranges
* Data Type Declaration Defaults
* Rules for Identifiers
* Special Values in VB
* VB Intrinsic Functions
* More on the Format Function
* Hungarian Notation

# Data Type Ranges

One of the first things that you must do when learning a new programming language is to determine what types of data can be natively represented in the language. Visual Basic has an extensive selection of built in data types. The following table lists the name of each data type, the purpose of each data type and how many bytes each type requires.

|  |  |  |
| --- | --- | --- |
| *Type* | *Description* | *Size in Bytes* |
| Boolean | Holds the value True or False | 2 |
| Byte | Holds values in the range 0 to 255 | 1 |
| Char | Holds any Unicode character (values 0 to 65,535) | 2 |
| Date | Holds any date in the range 1/1/1 to 12/31/9999 | 8 |
| Decimal | Holds real numbers; +/-79,228,162,514,264,337,593,950,335 with no decimal point or  +/-7.9228162512264337593543950335 x 1028 | 12 |
| Double | Holds floating point numbers using 64 bits in the range +/- 1.797693134862231 x 10308 | 8 |
| Integer | Holds an integer value in the range +/-2,147,483,648 (aliases to System.Int32) | 4 |
| Long | Holds an integer value in the range +/-9,223,372,036,854,775,808 (aliases to System.Int64) | 8 |
| Short | Holds an integer value in the range +/-32,768 (aliases to System.Int16) | 2 |
| Single | Holds floating point numbers using 32 bits in the range –1.401298x 10-45 to +3.402823 x 1038 | 4 |
| String | Holds groups of characters |  |
| Object | The ancestor of all other objects; can be used to point to other object types |  |

VB allows a programmer to specify a literal value’s data type by placing a character extension that is used to specify a data type. The following table lists the literal value data type extensions.

|  |  |  |
| --- | --- | --- |
| *Data Type* | *Format*  *Character* | *Example* |
| Char | C | "T"C |
| Date | # | #12/31/99# (notice two pound sign delimiters are required) |
| Decimal | D | 34D |
| Double | R | 400.55R |
| Integer | I | 300I (if you leave the extension off, it defaults to integer) |
| Long | L | 157L |
| Short | S | 3S |
| Single | F | 4.1F (F for floating point, since S was already used) |
| String | " | "Hi there" (notice that two delimiters are required here as well) |

# Data Type Declaration Defaults

VB now automatically initializes all variables as you declare them. The following table shows the default values for each data type

|  |  |
| --- | --- |
| *Data Type* | *Default Value* |
| Any Numeric Type (integers, bytes, longs, decimals, singles …) | 0 or 0.0 as appropriate |
| Boolean | False |
| Char | Char 0 (usually represented in hexadecimal as 0x0000) |
| Date | #01/01/0001 12:00:00AM# |
| String | Nothing (more on this in a moment) |

# Rules for Identifiers

You may have noticed that each constant had to be given a unique name. This name is formally known as an identifier. There are rules that VB imposes as to what can be used to make up an identifer. Here are those rules:

* No space is permitted in the identifier.
* The identifier must start with an alphabetic letter or underscore.
* You may use the underscore character in identifiers.
* The identifier cannot be the same as any VB keyword.
* Identifiers are not case sensitive.
* Identifiers may not exceed 255 characters in length.

You will use identifiers in many places in VB. For example, the name of a variable, the name of a constant, the name of a class, the name of a form, the name of subprogram, the name of a user defined data type and so on. As you can see, the concept of identifiers has a lot of importance to us in our programming life. As such, you should carefully pick good descriptive names for every item that you can name.

# Special Values in VB

We need to pause for a moment and examine some special values/keywords that VB allows us to use. While you may not have an immediate need for these items, they will eventually become quite important to you. This is an appropriate place to introduce these items since we are talking about data types.

**Nothing** – Indicates that an object variable has been declared but not yet assigned a value. Nothing is not the same as the empty string "" (a zero-length string). Here’s a slice of code that shows the IsNothing function being used to determine whether or not an object variable has been initialized and is referencing an object item or not:

If IsNothing(MyObjVar) Then

MessageBox.Show("Variable has not been initialized")

End If

You can also reset object variables to their empty (non-referencing) state by assigning them the Nothing value as in:

myObjVar = Nothing

**Null** – Indicates that a database field doesn’t contain any data. Null is not the same as Nothing, so make sure that you understand the difference between the two and that you don’t try to simply interchange them. You can test for this in the following manner:

If Not IsDBNull(SomeDBFieldVar) Then

'Since the variable isn't Null, do some

'processing on the variable

End If

# VB Intrinsic Functions

Here is a partial list of functions that are built into VB. You can assign the results from most of these functions into variables. We will cover some of these in more depth later:

|  |  |
| --- | --- |
| *Constant Name* | *Purpose* |
| E | Napier's constant 2.7182818… |
| Pi | Ratio of the circumference of a circle to its diameter 3.14159265… |

|  |  |
| --- | --- |
| *Math Namespace*  *Function* | *Purpose* |
| Abs(x) | returns the absolute value of numeric *x* |
| Acos(x) | returns the arccosine of *x* |
| Asin(x) | returns the arcsine of *x* |
| Atan(x) | returns the arctangent of *x* |
| Atan2(y,x) | returns the arctangent of *y*/*x* |
| BigMul(x,y) | returns the full product of two 32-bit numbers |
| Ceiling(x) | returns the smallest whole number greater than or equal to the specified number |
| Cos(x) | returns the cosine of *x* in radians |
| Cosh(x) | returns the hyperbolic cosine of *x* |
| DivRem(y,x,z) | returns the quotient of two 32 (or 64) bit integers and also returns the remainder in *z* |
| Exp(x) | returns the value of ex where e is Napier’s constant (2.71828…) |
| Floor(x) | returns the largest whole number less than or equal to the specified number |
| IEEERemainder(x,y) | returns the quotient of *x* /*y r*ounded to the nearest integer |
| Log(x) | returns the natural logarithm of *x* |
| Log10(x) | returns the base 10 logarithm of *x* |
| Max(x,y) | returns the larger of *x* or *y* |
| Min(x,y) | returns the smaller of *x* or *y* |
| Pow(x,y) | returns *x* raised to the *y* power |
| Round(x) | returns the integer value nearest to *x* |
| Sign(x) | returns a 1, 0 or –1 representing if *x* is positive, zero or negative |
| Sin(x) | returns the sin of *x* in radians |
| Sinh(x) | returns the hyperbolic sine of *x* |
| Sqrt(x) | returns the square root of *x* |
| Tan(x) | returns the tangent of *x* in radians |
| Tanh(x) | returns the hyperbolic tangent of *x* |
| Truncate(x) | returns the integer portion of *x* |

|  |  |
| --- | --- |
| *Function* | *Purpose* |
| Asc(x) | returns the ASCII value of character *x* |
| Chr(x) | prints the character associated with ASCII value *x* |
| Fix(x) | returns the integer portion of a number; if the number is negative, Fix will return the first negative integer greater than or equal to *x* |
| Hex(x) | returns the hexadecimal equivalent of numeric *x* |
| InStr(str1,str2) | tests if *str2* is a substring of *str1*: returns 0 if false, positive if true, which is the character position where *str2* appears in *str1* |
| InStrRev(str1,str2,[startat])) | just like InStr, but looks for last occurrence of a substring starting at the end of *str1* |
| Int(x) | returns the integer portion of a number; if the number is negative, Int will return the first negative integer less than or equal to *x* |
| IsArray(var) | returns True if the variable is an array |
| IsDate(str) | returns True if *str* is a date, False if *str* is not a date |
| IsDBNull(var) | returns True if the database *var* value is NULL, otherwise returns False |
| IsError(expr) | returns True if the expression is an exception type |
| IsNothing(var) | returns True if the object *var* points to nothing |
| IsNumeric(x) | returns True if *x* is a number, False if *x* is not a number |
| IsReference(var) | returns True if the *var* holds a reference to a valid object |
| Join | combines strings together with a specified delimiter |
| LBound(arrayvar) | returns the lower boundary of an array |
| LCase(str) | converts *str* to lowercase |
| Left(str,x) | returns the *x* leftmost characters of *str*  (Microsoft.VisualBasic. namespace) |
| Len(str) | returns the length of *str* |
| LSet(str,len) | left justifies the *str* to a length of *len* characters and pads spaces to the right |
| LTrim(str) | removes all spaces from the beginning of *str* |
| Mid(str,m,n) | returns *n* characters from *str* beginning with character *m* |
| MsgBox(param) | same as MessageBox.Show |
| Now() | returns the current serialized system date and time |
| Oct(x) | returns the octal value of numeric *x* |
| Randomize | initializes the random-number generator |
| Replace(str, whattofind, whattoputin) | returns a string that contains a version of *str* where all the *whattofind* have been replaced with *whattoputin* |
| Right(str,n) | returns the *n* rightmost characters from *str* (Microsoft.VisualBasic.namespace) |
| Rnd(x) | returns a random number |
| RSet(str, len) | right justifies the *str* to a length of *len* characters and pads spaces to the left |
| Rtrim(str) | removes all spaces from the right side of *str* |
| Space(x) | returns a string consisting of *x* spaces |
| Split | returns an array of strings split by a common delimiter |
| Str(x) | attempts to convert numeric *x* to a string |
| StrComp(s1, s2) | compares two strings and returns a result based on comparison |
| StrDup(x,char) | returns a string composed of *x* number of character *char* |
| StrReverse(str) | returns the reversed string |
| Trim(str) | removes all spaces at the beginning and end of *str* |
| TypeName(var) | returns a string containing the data type of a variable |
| UBound(arrayvar) | returns the upper boundary of an array |
| UCase(str) | converts *str* to all uppercase characters |
| Val(str) | attempts to convert *str* to a number |
| VarType(var) | returns an integer representing the data type of a variable |
| VbTypeName(var) | returns a string containing the Visual Basic data type name of a variable |

Just so you know: VB.Net treats Strings as a class type just like C++ and Java – thus strings are objects, not simple variables. Therefore, all strings have a built in number of methods that are handy to use. We will talk about the String class later on but for now you should at least get a feel for the capability of the string manipulations that are available.

## Converting Data from One Type to Another

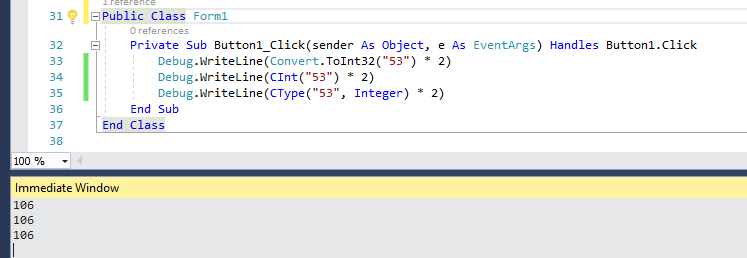
VB.NET provides two ways to perform data conversion activities. One style is to use the old VB conversion functions. These still work and are referenced in most .NET books, but again don’t be surprised if these functions are eventually removed from the language, however they have persisted this long…

|  |  |
| --- | --- |
| *Function* | *Purpose* |
| CBool() | Converts to a Boolean value |
| CByte() | Converts to a Byte value |
| CChar() | Converts to a Char value |
| CDate() | Converts to a Date value |
| CDbl() | Converts to a Double value |
| CDec() | Converts to a Decimal value |
| CInt() | Converts to an Integer value |
| CLng() | Converts to a Long value |
| CObj() | Converts to an Object reference |
| CSByte() | Converts to a Signed Byte value |
| CShort() | Converts to a Short value |
| CSng() | Converts to a Single value |
| CStr() | Converts to a String value |
| CUInt() | Converts to an Unsigned Integer value |
| CULng() | Converts to an Unsigned Long value |
| CUShort() | Converts to an Unsigned Short value |

The new style conversion methods come from the System.Convert namespace. Here are the methods available for your use:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Convert.ToBoolean() | Converts to a Boolean value |
| Convert.ToByte() | Converts to a Byte value |
| Convert.ToChar() | Converts to a Character value |
| Convert.ToDateTime() | Converts to a DateTime value |
| Convert.ToDecimal() | Converts to a Decimal value |
| Convert.ToDouble() | Converts to a Double value |
| Convert.ToInt16() | Converts to a Short value |
| Convert.ToInt32() | Converts to an Integer value |
| Convert.ToInt64() | Converts to a Long value |
| Convert.ToSByte() | Converts to a signed byte value |
| Convert.ToSingle() | Converts to a Single value |
| Convert.ToString() | Converts to a String value |
| Convert.ToUInt16() | Converts to an unsigned Short value |
| Convert.ToUInt32() | Converts to an unsigned Integer value |
| Convert.ToUInt64() | Converts to an unsigned Long value |

One other conversion choice that you have is the CType() function that allows you to specify the original value and the type you want it converted to. The syntax is: CType(*value*, *type\_to\_convert\_value\_to*). Here's a screenshot showing an example of the different conversion routines at work. Notice that all three of the conversion statements accomplish the same work by examining the output in the Immediate window:



# More on the Format Function

The Format function is used to display numbers and dates in a format that we want. Syntactically, the format function returns a string; the function itself is of the form Format(x, fmt). Here are some examples of the function in use:

'Use General Format

Debug.WriteLine(Format(Now(), "Long Date"))

Debug.WriteLine(Format(Now(), "MMM d yyyy"))

Debug.WriteLine(Format(12345.678, "Standard"))

Debug.WriteLine(Format(12345.678, "Currency"))

Debug.WriteLine(Format(12345.678, "#,0"))

Debug.WriteLine(Format(12345.678, "Percent"))

Debug.WriteLine(Format(12345.678, "Scientific"))

Debug.WriteLine(Format(1 / 4, "Standard"))

Debug.WriteLine(Format(-1234, "Currency"))

Listed below is the output generated by the Format lines sent to the Debug window:

Tuesday, December 19, 2017

Dec 19 2017

12,345.68

$12,345.68

12,346

1234567.80%

1.23E+04

0.25

($1,234.00)

There are many formatting characters available for use. Rather than rewriting all of this information, I reproduced the information directly from the Visual Studio MSDN Documentation.

*User-Defined Numeric Formats:*

The following table identifies characters you can use to create user-defined number formats. These may be used to build the style argument for the format function:

|  |  |
| --- | --- |
| ***Character*** | ***Description*** |
| None | Displays the number with no formatting. |
| (**0**) | Digit placeholder.Displays a digit or a zero. If the expression has a digit in the position where the zero appears in the format string, display it; otherwise, displays a zero in that position.  If the number has fewer digits than there are zeros (on either side of the decimal) in the format expression, displays leading or trailing zeros. If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal separator in the format expression, rounds the number to as many decimal places as there are zeros. If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, displays the extra digits without modification. |
| (**#**) | Digit placeholder.Displays a digit or nothing. If the expression has a digit in the position where the **#** character appears in the format string, displays it; otherwise, displays nothing in that position.  This symbol works like the **0** digit placeholder, except that leading and trailing zeros aren't displayed if the number has fewer digits than there are **#** characters on either side of the decimal separator in the format expression. |
| (**.**) | Decimal placeholder. The decimal placeholder determines how many digits are displayed to the left and right of the decimal separator. If the format expression contains only **#** characters to the left of this symbol; numbers smaller than 1 begin with a decimal separator. To display a leading zero displayed with fractional numbers, use zero as the first digit placeholder to the left of the decimal separator. In some locales, a comma is used as the decimal separator. The actual character used as a decimal placeholder in the formatted output depends on the number format recognized by your system. Thus, You should use the period as the decimal placeholder in your formats even if you are in a locale that uses a comma as a decimal placeholder. The formatted string will appear in the format correct for the locale. |
| (**%)** | Percent placeholder. Multiplies the expression by 100. The percent character (**%**) is inserted in the position where it appears in the format string. |
| (**,**) | Thousand separator. The thousand separator separates thousands from hundreds within a number that has four or more places to the left of the decimal separator. Standard use of the thousand separator is specified if the format contains a thousand separator surrounded by digit placeholders (**0** or **#**). A thousand separator immediately to the left of the decimal separator (whether or not a decimal is specified) or as the rightmost character in the string means "scale the number by dividing it by 1,000, rounding as needed."  For example, you can use the format string "##0,." to represent 100 million as 100,000. Numbers smaller than 1,000 but greater or equal to 500 are displayed as 1, and numbers smaller than 500 are displayed as 0. Two adjacent thousand separators in this position scale by a factor of 1 million, and an additional factor of 1,000 for each additional separator.  Multiple separators in any position other than immediately to the left of the decimal separator or the rightmost position in the string are treated simply as specifying the use of a thousand separator. In some locales, a period is used as a thousand separator. The actual character used as the thousand separator in the formatted output depends on the Number Format recognized by your system. Thus, You should use the comma as the thousand separator in your formats even if you are in a locale that uses a period as a thousand separator. The formatted string will appear in the format correct for the locale. |
| (**:**) | Time separator. In some locales, other characters may be used to represent the time separator. The time separator separates hours, minutes, and seconds when time values are formatted. The actual character used as the time separator in formatted output is determined by your system settings. |
| (**/**) | Date separator. In some locales, other characters may be used to represent the date separator. The date separator separates the day, month, and year when date values are formatted. The actual character used as the date separator in formatted output is determined by your system settings. |
| (**E- E+ e- e+**) | Scientific format. If the format expression contains at least one digit placeholder (**0** or **#**) to the left of **E-**, **E+**, **e-**, or **e+**, the number is displayed in scientific format and E or e is inserted between the number and its exponent. The number of digit placeholders to the left determines the number of digits in the exponent. Use **E-** or **e-** to place a minus sign next to negative exponents. Use **E+** or **e+** to place a minus sign next to negative exponents and a plus sign next to positive exponents. You must also include digit placeholders to the right of this symbol to get correct formatting. |
| **- + $** () | Literal characters. These characters are displayed exactly as typed in the format string. To display a character other than one of those listed, precede it with a backslash (**\**) or enclose it in double quotation marks (" "). |
| (**\**) | Displays the next character in the format string. To display a character that has special meaning as a literal character, precede it with a backslash (**\**). The backslash itself isn't displayed. Using a backslash is the same as enclosing the next character in double quotation marks. To display a backslash, use two backslashes (**\\**).  Examples of characters that can't be displayed as literal characters are the date-formatting and time-formatting characters (**a**, **c**, **d**, **h**, **m**, **n**, **p**, **q**, **s**, **t**, **w**, **y**, **/**, and **:**), the numeric-formatting characters (**#**, **0**, **%**, **E**, **e**, comma, and period), and the string-formatting characters (**@**, **&**, **<**, **>**, and **!**). |
| (**"ABC"**) | Displays the string inside the double quotation marks (" "). To include a string in the style argument from within code, you must use **Chr(34)** to enclose the text (**34** is the character code for a quotation mark (")). |

Format examples:

|  |  |  |  |
| --- | --- | --- | --- |
| *Format (Style)* | *"5" formatted as* | *"-5" formatted as* | *"0.5" formatted as* |
| Zero-length string ("") | 5 | -5 | 0.5 |
| 0 | 5 | -5 | 1 |
| 0.00 | 5.00 | -5.00 | 0.50 |
| #,##0 | 5 | -5 | 1 |
| $#,##0;($#,##0) | $5 | ($5) | $1 |
| $#,##0.00;($#,##0.00) | $5.00 | ($5.00) | $0.50 |
| 0% | 500% | -500% | 50% |
| 0.00% | 500.00% | -500.00% | 50.00% |
| 0.00E+00 | 5.00E+00 | -5.00E+00 | 5.00E-01 |
| 0.00E-00 | 5.00E00 | -5.00E00 | 5.00E-01 |

*Predefined numeric formats:*

|  |  |
| --- | --- |
| *Format name* | *Description* |
| **General Number**, **G**,or **g** | Displays number with no thousand separator. |
| **Currency**, **C**,or **c** | Displays number with thousand separator, if appropriate; display two digits to the right of the decimal separator. Output is based on system locale settings. |
| **Fixed**, **F**, or **f** | Displays at least one digit to the left and two digits to the right of the decimal separator. |
| **Standard**, **N**, or **n** | Displays number with thousand separator, at least one digit to the left and two digits to the right of the decimal separator. |
| **Percent**, **P**, or **p** | Displays number multiplied by 100 with a percent sign (%) appended to the right; always display two digits to the right of the decimal separator. |
| **Scientific**, **E**, or **e** | Uses standard scientific notation. |
| **D**, or **d** | Displays number as a string that contains the value of the number in Decimal (base 10) format. This option is supported for integral types **(Byte**, **Short**, **Integer**, **Long**) only. |
| **X**, or **x** | Displays number as a string that contains the value of the number in Hexadecimal (base 16) format. This option is supported for integral types (**Byte**, **Short**, **Integer**, **Long**) only. |
| **Yes/No** | Displays No if number is 0; otherwise, displays Yes. |
| **True/False** | Displays False if number is 0; otherwise, displays True. |
| **On/Off** | Displays Off if number is 0; otherwise, displays On. |

*Predefined date/time formats:*

|  |  |
| --- | --- |
| *Format Name* | *Description* |
| **General Date**, or **G** | Displays a date and/or time. For real numbers, display a date and time; for example, 4/3/93 05:34 PM.If there is no fractional part, display only a date, for example, 4/3/93. If there is no integer part, display time only, for example, 05:34 PM. Date display is determined by your system's **LocaleID** value. |
| **Long Date**, or **D** | Displays a date according to your locale's long date format. |
| **Medium Date** | Displays a date using the medium date format appropriate for the language version of the host application. |
| **Short Date**, or **d** | Displays a date using your locale's short date format. |
| **Long Time**, or **T** | Displays a time using your locale's long time format; includes hours, minutes, seconds. |
| **Medium Time** | Displays time in 12-hour format using hours and minutes and the AM/PM designator. |
| **Short Time**, or **t** | Displays a time using the 24-hour format, for example, 17:45. |
| **F** | Displays the long date and short time according to your locale's format. |
| **F** | Displays the long date and long time according to your locale's format. |
| **G** | Displays the short date and short time according to your locale's format. |
| **M**, **m** | Displays the month and the day of a date. |
| **R**, **r** | Formats the date and time as Greenwich Mean Time (GMT) |
| **S** | Formats the date and time as a sortable index. |
| **U** | Formats the date and time as a GMT sortable index. |
| **U** | Formats the date and time with the long date and long time as GMT. |
| **Y** | Formats the date as the year and month. |

*User-defined date/time formats:*

|  |  |
| --- | --- |
| *Character* | *Description* |
| (**:**) | Time separator. In some locales, other characters may be used to represent the time separator. The time separator separates hours, minutes, and seconds when time values are formatted. The actual character used as the time separator in formatted output is determined by your system's **LocaleID** value. |
| (**/**) | Date separator. In some locales, other characters may be used to represent the date separator. The date separator separates the day, month, and year when date values are formatted. The actual character used as the date separator in formatted output is determined by your locale. |
| (**%**) | Used to indicate that the following character should be read as a single-letter format without regard to any trailing letters. Also used to indicate that a single-letter format is read as a user-defined format. See below for further details |
| **D** | Displays the day as a number without a leading zero (for example, 1). Use **%d** if this is the only character in your user-defined numeric format. |
| **Dd** | Displays the day as a number with a leading zero (for example, 01). |
| **Ddd** | Displays the day as an abbreviation (for example, Sun). |
| **Dddd** | Displays the day as a full name (for example, Sunday). |
| **M** | Displays the month as a number without a leading zero (for example, January is represented as 1). Use **%M** if this is the only character in your user-defined numeric format. |
| **MM** | Displays the month as a number with a leading zero (for example, 01/12/01). |
| **MMM** | Displays the month as an abbreviation (for example, Jan). |
| **MMMM** | Displays the month as a full month name (for example, January). |
| **Gg** | Displays the period/era string (for example, A.D.) |
| **H** | Displays the hour as a number without leading zeros using the 12-hour clock (for example, 1:15:15 PM). Use **%h** if this is the only character in your user-defined numeric format. |
| **Hh** | Displays the hour as a number with leading zeros using the 12-hour clock (for example, 01:15:15 PM). |
| **H** | Displays the hour as a number without leading zeros using the 24-hour clock (for example, 1:15:15). Use **%H** if this is the only character in your user-defined numeric format. |
| **HH** | Displays the hour as a number with leading zeros using the 24-hour clock (for example, 01:15:15). |
| **M** | Displays the minute as a number without leading zeros (for example, 12:1:15). Use **%m** if this is the only character in your user-defined numeric format. |
| **Mm** | Displays the minute as a number with leading zeros (for example, 12:01:15). |
| **S** | Displays the second as a number without leading zeros (for example, 12:15:5). Use **%s** if this is the only character in your user-defined numeric format. |
| **Ss** | Displays the second as a number with leading zeros (for example, 12:15:05). |
| **F** | Displays fractions of seconds. For example **ff** will display hundreths of seconds, whereas **ffff** will display ten-thousandths of seconds. You may use up to seven **f** symbols in your user-defined format. Use **%f** if this is the only character in your user-defined numeric format. |
| **T** | Uses the 12-hour clock and displays an uppercase A for any hour before noon; displays an uppercase P for any hour between noon and 11:59 P.M. Use **%t** if this is the only character in your user-defined numeric format. |
| **Tt** | Uses the 12-hour clock and displays an uppercase AM with any hour before noon; displays an uppercase PM with any hour between noon and 11:59 P.M. |
| **Y** | Displays the year number (0-9) without leading zeros. Use **%y** if this is the only character in your user-defined numeric format. |
| **Yy** | Displays the year in two-digit numeric format with a leading zero, if applicable. |
| **yyy** | Displays the year in four digit numeric format. |
| **yyyy** | Displays the year in four digit numeric format. |
| **Z** | Displays the timezone offset without a leading zero (for example, -8). Use **%z** if this is the only character in your user-defined numeric format. |
| **Zz** | Displays the timezone offset with a leading zero (for example, -08) |
| **zzz** | Displays the full timezone offset (for example, -08:00) |

Example -- The following are examples of user-defined date and time formats for December 7, 1958, 8:50 PM, 35 seconds.

|  |  |
| --- | --- |
| Format | Displays |
| M/d/yy | 12/7/58 |
| d-MMM | 7-Dec |
| d-MMMM-yy | 7-December-58 |
| d MMMM | 7 December |
| MMMM yy | December 58 |
| Hh:mm tt | 08:50 PM |
| h:mm:ss t | 8:50:35 P |
| H:mm | 20:50 |
| H:mm:ss | 20:50:35 |
| M/d/yyyy H:mm | 12/7/1958 20:50 |

## Format Related Classes

Visual Basic.Net also provides you with some classes that you can use for formatting. We previously examined these in the second chapter.

FormatCurrency(*number*, *#OfDigitsAfterDecimal*, *IncludeLeadingDigit*,

*UseParensForNegativeNumbers*, *GroupDigits*)

FormatDateTime(*datetimeValue*, *NamedFormat*)

FormatPercent(*number*, *#OfDigitsAfterDecimal*, *IncludeLeadingDigit*,

*UseParensForNegativeNumbers*, *GroupDigits*)

FormatNumber(*number*, *#OfDigitsAfterDecimal*, *IncludeLeadingDigit*,

*UseParensForNegativeNumbers*, *GroupDigits*)

## One More Side Trip on Formatting: String.Format

Microsoft has provided one more method for formatting information to make it look nice when it’s output: the String.Format method. This method allows placeholders and format descriptors to be placed in a string that will specify how the data is to actually look when it is inserted. The following table provides a summary of the special formatting characters (either case letter can be used). You can think of this as a more modern version of the Format function we examined just a few moments ago.

String.Format descriptors:

|  |  |
| --- | --- |
| *Descriptor* | *Purpose* |
| C | Currency |
| D | Decimal |
| E | Exponential |
| F | Fixed-point |
| G | General |
| N | Number |
| P | Percentage |
| R | Roundtrip – ensures that numbers converted to strings will get the same value when they are converted back to numbers |
| X | Hexadecimal |
| 0 | Indicates where a digit must be printed |
| # | Allows suppression of leading digits |
| . | Allows the positioning of the decimal |
| , | Allows the positioning of the group separator |
| % | Allows the positioning of the decimal place |

It is important to note that the placeholders are assigned sequentially, left to right, starting with value {0}. Any formatting descriptors will follow the placeholder number and a colon. Here are some examples of the String.Format method put to work on simple numbers:

Debug.WriteLine(String.Format("Your Answer is {0} which was " & \_

"{1} divided by {2}", \_

(3.0 / 4.0), 3, 4))

Debug.WriteLine("C descriptor examples")

Debug.WriteLine(String.Format("My number {0:C}", 1234.567))

Debug.WriteLine(String.Format("My number {0:c}", -1234.567))

Debug.WriteLine("D descriptor examples")

Debug.WriteLine(String.Format("{0:D}", 1234))

Debug.WriteLine(String.Format("{0:D6}", 1234))

Debug.WriteLine("E descriptor examples")

Debug.WriteLine(String.Format("{0:E}", 123.4))

Debug.WriteLine(String.Format("{0:E6}", 123.4))

Debug.WriteLine(String.Format("{0:E4}", 123.4))

Debug.WriteLine("F descriptor examples")

Debug.WriteLine(String.Format("{0:F} {1:F7} {2:F0}", \_

123.45, 123.45, 123.45))

Debug.WriteLine("G descriptor examples")

Debug.WriteLine(String.Format("{0:G}", 12345.67))

Debug.WriteLine(String.Format("{0:G4}", 12345.67))

Debug.WriteLine(String.Format("{0:G6}", 12345.67))

Debug.WriteLine("N descriptor examples")

Debug.WriteLine(String.Format("{0:N} {1:N3}", 12345.67, 12345.67))

Debug.WriteLine("X descriptor examples")

Debug.WriteLine(String.Format("{0:x4}", 65))

Debug.WriteLine("0 descriptor examples")

Debug.WriteLine(String.Format("{0:00}", 12))

Debug.WriteLine(String.Format("{0:0000}", 123))

Debug.WriteLine("# descriptor examples")

Debug.WriteLine(String.Format("{0:####}", 123))

Debug.WriteLine("#. descriptor examples")

Debug.WriteLine(String.Format("{0:####.000}", 123456.7))

Debug.WriteLine(String.Format("{0:##.000}", 12345.67))

Debug.WriteLine(String.Format("{0:#.000}", 1.234567))

Debug.WriteLine("#, descriptor examples")

Debug.WriteLine(String.Format("{0:##,###}", 123456.7))

Debug.WriteLine(String.Format("{0:##,###,000.000}", 1234567.123456))

Debug.WriteLine(String.Format("{0:#,#.00}", 1234567.1234567))

Debug.WriteLine("% descriptor examples")

Debug.WriteLine(String.Format("{0:##,000%}", 123.45))

Debug.WriteLine(String.Format("{0:00%}", 0.123))

Here's the output from the lines of code above:

Your Answer is 0.75 which was 3 divided by 4

C descriptor examples

My number $1,234.57

My number ($1,234.57)

D descriptor examples

1234

001234

E descriptor examples

1.234000E+002

1.234000E+002

1.2340E+002

F descriptor examples

123.45 123.4500000 123

G descriptor examples

12345.67

1.235E+04

12345.7

N descriptor examples

12,345.67 12,345.670

X descriptor examples

0041

0 descriptor examples

12

0123

# descriptor examples

123

#. descriptor examples

123456.700

12345.670

1.235

#, descriptor examples

123,457

1,234,567.123

1,234,567.12

% descriptor examples

12,345%

12%

While we only looked at numbers in the String.Format example above, realize that there are also format strings for Date/Times and other types as well. We introduced some of these back in chapter two. In summary, with all of the various formatting options available to you, it should be very little work to make all of your data look nice. There really is no excuse in VB to accept anything less. In fact, if you don’t take the time and effort to line things up nice and neat, I won’t take the time or effort to give you all of your points. You’ve been warned! Crappy output is not acceptable…

# Hungarian Notation

This section will list the various Hungarian Notation naming conventions that Microsoft used to suggest for use. As mentioned early, you should follow this coding convention to help make your code more readable. All that you need to do is to prefix the identifier name of any item you create with the three letter prefix that depicts the data type that the item is.

This is not an exhaustive list of all possible prefixes. I only provided the prefixes for items that I expect you will be using.

**Control prefixes:**

|  |  |  |
| --- | --- | --- |
| *Control type* | *Prefix* | *Example* |
| Check box | chk | chkAlwaysSaveFile |
| Combo box | cbo | cboIceCreamType |
| Command button | cmd | cmdExit |
| Form | frm | frmEmployeeHire |
| Horizontal scroll bar | hsb | hsbVolume |
| Label | lbl | lblEnterName |
| Listbox | lst | lstStates |
| ListView | lvw | lvwTopics |
| MDI child form | mdi | mdiWorkspace |
| Menu | mnu | mnuFileOpen |
| Picture | pic | picEmployeePicture |
| ProgressBar | prg | prgLoadFile |
| RichTextBox | rtb | rtbOutput |
| Slider | sld | sldScale |
| StatusBar | sta | staProgOptions |
| TabStrip | tab | tabOptionsPage |
| Text box | txt | txtUsername |
| Timer | tmr | tmrAlarm |
| Toolbar | tlb | tlbMainAppToolbar |
| TreeView | tre | treDirectories |
| UpDown | upd | updDirection |
| Vertical scroll bar | vsb | vsbRate |

**Data type prefixes:**

|  |  |  |
| --- | --- | --- |
| *Data type* | *Prefix* | *Example* |
| Boolean | bln | blnProgramDone |
| Byte | byt | bytStatus |
| Char | chr | chrMiddleInitial |
| Collection | col | colEmployees |
| DateTime | dtm | dtmHireDate |
| Decimal | dec | decTemperature |
| Double | dbl | dblSolution |
| Integer | int | intNumberOfStudents |
| Long | lng | lngTotalQuantity |
| Object | obj | objCar |
| Short | sht | shtCount |
| Single | sng | sngSalary |
| String | str | strFirstName |
| User-defined type | udt | udtMachine |

**Variable scope prefixes:**

|  |  |  |
| --- | --- | --- |
| Scope | Prefix | Example |
| Global | g | gintTotalItemsProduced |
| Module-level | m | mstrReportName |
| Local to procedure | None | dblSalary |

Each word in a variable should be Pascal cased after the Hungarian Notation, e.g. intCountofStudentCreditHours, strFirstName, sngGradePointAverage.

**Constant Naming Conventions :**

Constants typically are written all uppercase letters. In addition, placing the standard variable type and scope prefixes in lowercase at the beginning of the constant name helps in the readability and understanding of the constant’s purpose and type. Examples: sngMI\_SALES\_TAX\_RATE = 0.04, intSCREEN\_COLUMNS = 80, chrRUN\_AGAIN = “Y”

**Subprogram Names:**

All subprogram names will be verb based/action words, except for those functions whose purpose is understandable, e.g. the built-in functions like sqrt() and sin(). Some subprogram examples include GenerateReport, ReadDataFile, and CalculateMean.

**Suggested Prefixes for Menus:**

Since the default names given by the menu builder in the Visual Studio IDE are terrible (MenuItem1 for example), we should name each item following the menu hierarchy that it is implementing. Some examples include:

File🡪Open mnuFileOpen

File🡪Print🡪PDF mnuFilePrintPDF

Format🡪Character mnuFormatCharacter

Help🡪About mnuHelpAbout

**Order of various program items/sections:**

All programs will have their various items appear in the following order: constants, classes, type definitions and variables. This will be true for each appropriate scope level.

As you can see, simply adding a few extra letters up front to your identifiers can greatly increase the information that you are conveying. By coupling Hungarian Notation with the practice of selecting good names for your variables, constants and controls, you can simplify the amount of work it takes to understand the code you have written. In addition, the concern of having to do extra typing is greatly diminished by the fact that the IntelliSense system in VB.NET will allow you to complete the names of identifiers with very few letters being typed on your part.