Chapter 2 – Every Language Has Rules

In the previous chapter, you saw some simple InternetBasic programs. Before we move to some of the more advanced IB features, let's stop and talk about some of the rules. Here are some of the major ones (with some others to follow in later chapters):

IB source programs are text files

Did we forget to mention that in Chapter 1? Oops, sorry about that. Okay, now we've covered the most basic rule of all.

IB source programs consist of two sections

Sure, we said this in Chapter 1, but this one's so important that's it's worth repeating. The first section of an IB source program contains definitions of the data for the program, such as variable names, sizes, and types (and other features that we'll talk about in future chapters). The second section of the source program contains the executable code.

Statements are not case sensitive

You may have already noticed this, but IB statements, variable names, and statement labels may be written in upper or lower case (or a combination thereof).

Comment character

The exclamation mark (!) is the comment character in IB source programs. Whenever it appears on a source line, the IB compiler ignores the remainder of that line. (The only exception to this rule is when an exclamation mark appears inside of quotation marks, in which case the IB compiler treats as part of a character string.)

Compiler scripting command

You can include an IB compiler scripting command at the beginning of each source program. This command tells the IB compiler about the source program and object program. Here's an example:

```
! //MTB// Src(SAMPLE1.MTB,BAS) Obj(SAMPLE1,BAS)
```

Multiple statements on a line

One of the many convenient programming features of IB is the ability to write more than one source statement on the same line. A special character is required for this purpose; the character is the ampersand (&).

The ampersand permits multiple statements to be written on a single line. Please note that IB supports source lines of up to 132 characters each.

The ampersand character can be used in all sections of an IB source program. Here are some examples:

Example 1:

```
A=10 & B=20 & C=30
```

This example shows how three assignment statements can be written on a single source line.

Example 2:

```
LENGTH 8.2 & LOCAL CRLIMIT
```

This example shows how two data statements (LENGTH and LOCAL) can be written on a single source line.

Statement labels

Many IB statements may begin with a statement label. In the data section of the program, the FORMAT statement (see Chapter 3 for more information) contains a label. In the code section of the program, any statement may begin with a statement label (and, in fact, a statement label may appear on a line by itself).

Statement labels may be alphanumeric or numeric.

Alphanumeric labels must begin with a letter and end with a colon (:) when being defined (other references do not include the colon). Characters following the first one may be alphabetic or numeric. No special symbols or blanks are allowed in statement labels. Alphanumeric labels may be any length, but the first 32 characters must be unique within a single source program.

Numeric labels may contain up to four digits (e.g., 1, 10, 100, 1000). The IB compiler treats numeric labels as a string of characters, rather than as a numeric value. This means that labels such as 1, 01, 001, and 0001 are all different from each other.

Example:

```
Label        Statement

Top:         READ (1,CUSTFILE) EXCP=ReadError
             PRINT (2) CUSTOMERNAME$;CUSTOMERNUMBER$
             .
             .
             GOTO Top

ReadError:   ! This routine determines the error
             .
             .
             .
```

This example demonstrates two statement labels. The first line of code contains the label Top:, which is the first line of a loop structure. Later in the code, the GOTO statement sends control back to the Top statement.

The second statement label is ReadError, which marks the start of a routine that determines what type of runtime error might occur in this program. Notice that the statement label is referred to in the EXCP= parameter of the READ statement. (See Chapter ??? for more information about the READ statement.)

Statement continuation

Sometimes, you may find it desirable to continue a source statement on the following editor line. This is done with an underscore (_) placed at the end of a source line.

The underscore signals the IB compiler that the current statement is being continued on the following source line.

There is no limit to the number of lines that may be joined using the underscore.

Note: The underscore can be used in all sections of an IB source program.

The IB compiler ignores any characters following the underscore. These superfluous characters are treated as remarks and not compiled into the object program.

Example:

```
Print _
"Choose one of the following programs",@(0,3);_
"-----------------------------------",@(0,4);_
"1 -- Temperature conversion",@(0,6);_
"2 -- Break even calculation",@(0,8);_
"3 -- Number guessing game",@(0,10);_
"4 -- Coin counting example",@(0,12);_
"5 -- Fibonacci numbers and the 'Golden Mean'",@(0,14);_
"6 -- What happens if you double your money every day?",@(0,16)
```

In this example, a single Print statement displays the entire menu of selections. Notice that the underscore is used on each line except for the final one. Also notice that the semicolon (;) is used to separate the print items in this Print statement.

What if you want to use an underscore as part of your program (in a text string, for example)? Simple, just use the “escape” character (the @ symbol) immediately before the underscore, in which case the compiler will not treat it as a line continuation character, but will display the underscore character.

Example:

```
Print "This is the underscore @_ character",@(0,3);_
```

Indentation

IB statements may start at any position on the line. For more readable source code, you may indent instructions in order to highlight statement labels, illustrate loop constructs, show nested structures, etc.

Example 1:

Before indentation

```
FOR QUARTERS=0 TO 4
FOR DIMES=0 TO 10
FOR NICKELS=0 TO 20
TOTAL = NICKELS*.05 + DIMES*.1 + QUARTERS*.25
IF TOTAL = 1 THEN
ROW = ROW + 1
PRINT NICKELS,@(0,ROW);DIMES,@(10,ROW);QUARTERS,@(20,ROW)
COUNTER = COUNTER + 1
ENDIF
NEXT NICKELS
NEXT DIMES
NEXT QUARTERS
```

After indentation

```
FOR QUARTERS=0 TO 4
  FOR DIMES=0 TO 10
    FOR NICKELS=0 TO 20
      TOTAL = NICKELS*.05 + DIMES*.1 + QUARTERS*.25
        IF TOTAL = 1 THEN
          ROW = ROW + 1
          PRINT NICKELS,@(0,ROW);DIMES,@(10,ROW);QUARTERS,@(20,ROW)
          COUNTER = COUNTER + 1
        ENDIF
    NEXT NICKELS
  NEXT DIMES
NEXT QUARTERS
```

Data types

IB supports two data types: string and numeric. String data may include alphanumeric characters, special characters, and hex values. Numeric data may include digits, a decimal point, and a minus sign only.

String data

String data may be classified as constant or variable. String constants (sometimes called string literals) are characters enclosed in quotation marks (single or double). String constants may contain up to 254 characters. For example, the following are considered string constants (note that hex constants require a leading and trailing @ character):

String constant	Explanation
“InternetBasic”	alphabetic characters
“4325 Harrison Grade Road”	alphanumeric characters
“(707) 874-1250”	alphanumeric and special characters
“@FF@”	hex constant FF
“@2020@”	hex constant 2020

String variables are named memory locations used to store string data (up to 254 characters per variable). In IB, string variable names may be any number of characters long, although the compiler recognizes only the first 30 characters of the name. String variable names may be upper or lower case, may include imbedded decimal points, and must end with a dollar sign ($) character. They may not include imbedded blanks or other special characters. For example, the following are valid string variable names:

```
CUSTOMER$
Customer$
Customer$
PARTNUMBER$
Part.number$
part.number$
A$
a$
VALUE1$
value1$
CITYSTATEZIPCODE$
City.State.ZIP$
```

Numeric data

Numeric data may be classified as constant or variable. Numeric constants are fixed numeric values included in a program; they may contain digits, a decimal point, and a trailing minus sign. Commas are not allowed in numeric constants (but may be added to the output via edit masks – see Chatper ??? for more information). Numeric constants may contain up to 16 digits, 15 of which may be located to the right side of the decimal point. For example, the following values are numeric constants:

```
100
1234.56
2500.9999-
1.234567890123456
1234567890123456.
```

Numeric variables are named memory locations used to store numeric data (up to 16 digits per variable). In IB, numeric variable names may be any number of characters long, although the compiler recognizes only the first 30 characters of the name. Numeric variable names may be upper or lower case and may include imbedded decimal points. They may not include imbedded blanks or other special characters. For example, the following are valid numeric variable names:

```
CREDITLIMIT
```

```
Credit.limit
AROVER120
AR.over.120
INVENTORYVALUE
inventory.value
X
X
AMOUNT
amount
```

As you've already seen in the sample programs in Chapter 1, all variables must be defined before they can be used. These definitions include the variable name, type (string or numeric), length, and classification.

LOCAL and COMMON data

Each variable must be classified as **local** or **common**. Local variables are those intended for a single program only. Common variables, on the other hand, are those to be shared with other IB programs (program overlays, subprograms, or IB programs running as background sessions). Therefore, each variable must be defined according to its intended purpose within the application.

Common variables must be defined in the same order and with the same length (and precision) in all programs that share the data. They do not have to have the same variable names in separate programs, though; they just need to be mapped in the same order and with the same size.

For example, here's how to define a local 10-character string variable named ZIPCODE$:

```
LENGTH 10
LOCAL ZIPCODE$
```

On the other hand, here's how to make ZIPCODE$ a common variable:

```
LENGTH 10
COMMON ZIPCODE$
```

System variables

IB includes several system variables. These are variables that are available to your program, but do not have to be defined in your source code. We will skip the details for now and introduce these system variables in subsequent chapters.

Arrays (subscripted variables)

InternetBasic supports arrays (subscripted variables) for both numeric and string variables. Arrays are defined via the LOCAL and COMMON statements in the data section of the program.

While there is no theoretical limit to the number of dimensions in an array, an array may contain a maximum of 32,768 bytes.

The following example shows how to define a one-dimensional string variable named **LIST$**. This array contains 200 elements, each with a length of 20 bytes):

```
LENGTH 20             ! Length of 20 bytes
LOCAL LIST$(200)      ! Array variable with 200 elements
```

In the code section, an array variable is referred to using a numeric constant or numeric variable as the array subscript. The first subscript is 1.

For example:

```
LIST$(1) = "HELLO"     ! numeric constant as subscript

or

FOR I = 1 TO 200
LIST$(I) = "HELLO"     ! numeric variable as subscript
NEXT I
```

Note: When a variable subscript is used, IB verifies that the calculated offset for the array element is within the total array space only. It does not check to see if the size of any particular dimension has been exceeded. If the calculated offset falls beyond the array space, IB issues an “array subscript out of range” error.

Next, we'll expand the **LIST$** variable to a two-dimensional variable:

```
LENGTH 20             ! Length of 20 bytes
LOCAL LIST$(200,2)    ! Array variable with 200 x 2 elements
```

Here's a three-dimensional array named **CHART$**, with 10 x 5 x 8 dimensions, each element defined as a 15-byte string:

```
LENGTH 15             ! Length of 15 bytes
LOCAL CHART$(10,5,8)  ! Array variable with 10 x 5 x 8 elements
```

Finally, here's six-dimensional numeric array named **DATA**, with 4 x 4 x 4 x 4 x 4 x 4 dimensions, each element defined as length/precision of 2.0:

```
LENGTH 2.0               ! Length of 2.0 digits
LOCAL DATA(4,4,4,4,4,4)  ! Array variable with 4 x 4 x 4 x 4 x 4 x 4
elements
```

Note: While all of the above examples show LOCAL variables, they could also be defined as COMMON variables.

Operations

InternetBasic supports numeric operations, string operations, relational operations, and logical operations. In addition to these operations, there are many functions that operate on string and/or numeric data. For information about these functions, see Chapter ???.

Addition

The plus sign (+) is the operator used to perform addition on numeric data.

Notes:

- If the result of an addition operation exceeds the defined length of a receiving numeric variable, the value is truncated on the left-hand side without warning.
- If the precision of the result exceeds the defined precision of a receiving variable (and the variable has not been defined for automatic rounding), the result is truncated on the right-hand side. See the ROUND statement (Chapter ???) for more information.

Example:

```
LENGTH 5.0                  ! Declare the length for A, B, C
LOCAL A, B, C               ! Declare them as LOCAL variables
.
LET A = 500                 ! Set the value for A
LET B = 2000                ! Set the value for B
LET C = A + B               ! Add A and B, giving C
```

Subtraction

The minus sign (-) is the operator used to perform subtraction on numeric data.

Notes:

- If the result of a subtraction operation exceeds the defined length of a receiving numeric variable, the value is truncated on the left-hand side without warning.
- If the precision of the result exceeds the defined precision of a receiving variable (and the variable has not been defined for automatic rounding), the result is truncated on the right-hand side. See the ROUND statement for more information.

Example:

```
LENGTH 5.0                  ! Declare the length for A, B, C
LOCAL A, B, C               ! Declare them as LOCAL variables
.
LET A = 800                 ! Set the value for A
LET B = 200                 ! Set the value for B
LET C = A - B               ! Subtract B from A, giving C
```

Multiplication

The asterisk (*) is the operator used to perform multiplication on numeric data.

Notes:

- If the result of a multiplication operation exceeds the defined length of a receiving numeric variable, the value is truncated on the left-hand side without warning.
- If the precision of the result exceeds the defined precision of a receiving variable (and the variable has not been defined for automatic rounding), the result is truncated on the right-hand side. See the ROUND statement for more information.

Example:

```
LENGTH 5.0                  ! Declare the length for A, B, C
LOCAL A, B, C               ! Declare them as LOCAL variables
.
LET A = 150                 ! Set the value for A
LET B = 25                  ! Set the value for B
LET C = A * B               ! Multiply A and B, giving C
```

Division

The backslash (/) is the operator used to perform division on numeric data.

Notes:

- If the result of a division operation exceeds the defined length of a receiving numeric variable, the value is truncated on the left-hand side without warning.
- If the precision of the result exceeds the defined precision of a receiving variable (and the variable has not been defined for automatic rounding), the result is truncated on the right-hand side. See the ROUND statement for more information.
- The result of a division operation is as precise as the most precise value in the formula.

Example:

```
LENGTH 5.0                  ! Declare the length for A, B, C
LOCAL A, B, C               ! Declare them as LOCAL variables
.
LET A = 150                 ! Set the value for A
LET B = 25                  ! Set the value for B
LET C = A / B               ! Divide A by B, giving C
```

Modulo

The MOD operator performs a modulo operation on the two numeric values (i.e., it divides the first value by the second value and returns the remainder). The syntax is:

```
numeric-value-1 MOD numeric-value-2
```

Example:

```
10 MOD 3
```

This operation returns a value of 1. Explanation: 10 divided by 3 equals 3 with a remainder of 1.

Precedence

In a numeric expression containing multiple operations, the multiplication and division operations are performed first (from left to right). Next, the addition and subtraction operations are performed (also from left to right).

This algebraic order can be overridden with parentheses. In a numeric expression containing parentheses, operations inside the parentheses are performed first, followed by the algebraic order previously described. Parentheses may be nested to any level in a numeric expression; they are performed from the inside to the outside.

Results

The temporary result of a numeric operation contains only as many digits to the right of the decimal point as the most precise operand in the expression. For example, if an expression contains operand with two digits of precision and four digits of precision, the temporary result contains four digits of precision.

String operations

IB supports the string operation of concatenation. The plus sign (+) is used to indicate concatenation (the combining of strings). For example, concatenating two string variables would look like this:

```
FIRSTNAME$ + LASTNAME$
```

You may concatenate string constants, string variables, and string functions. Here is an example of concatenating constants with variables:

```
"(" + AREACODE$ + ")" + PHONENUMBER$
```

Note: The maximum number of characters in any string is 254. If a concatenation operation results in more characters than have been declared for a receiving variable, surplus characters are truncated on the right-hand side. **This truncation occurs without warning of any kind.**

Relation operations

IB supports relational operations in IF/THEN statements. The following operations are supported:

```
Operation                       IB syntax
================================================================
Equal to                        EQ  or  =
Not equal to                    NE  or  NOT=
```

```
Greater than                    GT  or  >
Greater than or equal to        GE  or  >=
Less than                       LT  or  <
Less than or equal to           LE  or  <=
```

Additional information:

- When comparing string values, the ASCII values for the characters are compared. For example, upper case "A" is considered to be less than lower case "a" because the ASCII value for "A" (i.e., 65) is less than the ASCII value for "a" (i.e., 97). Thus, it is possible to determine correct alphabetical order for strings using relational comparisons. See the ASCII chart (Appendix ???).
- Two strings are considered to be equal only when they are identical and have the same current length (number of characters).

Logical operations

IB supports the AND and OR logical operations to be used in conjunction with relational expressions. The AND operator tests whether both relational conditions are true, while the OR operator tests for truth of either relational condition.

For example:

```
IF A=B AND B=C THEN...        !  Both conditions must be true

IF A=B OR B=C THEN...         !  Either condition may be true
```

If AND and OR operators are combined in a relational expression, the AND operator takes precedence (i.e., it is evaluated first). It is possible to override this precedence with parentheses, just like the precedence in numeric operations can be overridden.

Every language has reserved words

When composing your own InternetBasic programs, take care not to use reserved words for your own purposes such as variable names and statement labels. See Appendix ??? for a list of reserved words.

Every language has special characters

Here are the special characters that are used in InternetBasic.

!	The exclamation mark allows comments to appear on source lines without being compiled into the object program.
&	The ampersand permits multiple statements to be written on a single line.
_	The underline is used to continue a statement to the following source line.

+	The plus sign is used in numeric expression to indicate addition and in string expressions to indicate concatenation. It is also used in the edit mask portion of a FORMAT statement (see Chapter ???).
-	The minus sign is used in numeric expressions to indicate subtraction. It is also used in the edit mask portion of a FORMAT statement.
*	The asterisk is used in numeric expressions to indicate multiplication. It is also used in the edit mask portion of a FORMAT statement.
/	The forward slash is used in numeric expressions to indicate division.
.	The decimal point is used in numeric constants to indicate position of the decimal point. It is also used in the edit mask portion of a FORMAT statement.
,	The comma is used separate parameters in some IB statements. The command may be replaced with a blank space except in array references. It is also used in the edit mask portion of a FORMAT statement.
;	The semicolon separates items in input/output statements and FORMAT statements.
" or '	Double or single quotes enclose string constants (e.g., "YES", "QUIT", "SAVE"). These must be paired so that an opening double quote must be closed with a double quote (and same for single quotes). Either type of quote may appear inside a pair of the other type (e.g., "IT'S TIME.").
( )	Left and right parentheses enclosed various parameters in IB programs, including array subscripts and dimensions, function arguments, logical unit numbers, format labels (when referred to in the code section), cursor position coordinates, algebraic expressions, and edit masks. Left and right parentheses must be paired.
$	The dollar sign is used as the ending character in string variable names. It is also used as the leading character in an edit mask.
=	The equal sign is used to assign the results of an expression to a variable. It is also used in relational operations to compare two values for equality.
<	The less than symbol is used in relational operations to compare two values.
>	The greater than symbol is used in relational operations to compare two values.
@	The “at” sign is used to specify cursor position or print position in PRINT, INPUT, and FORMAT statements. It is also used to enclose hex constants (e.g., "@FF@"), and is also the “escape” character within a string constant (e.g., “This is an underscore @_ character.”)
#	The number sign (also know as the “pound” sign and "sharp" sign) serves various purposes. It is primarily used in edit masks, and may also be used to specify the user record buffer in I/O statements (see Chapter ??? for more information).

The INPUT statement strips trailing blanks from strings

The IB runtime system automatically strips trailing blanks from a string field immediately after it has been input. For example, suppose you have defined a string field named A$ with a length of 10 characters, and you use the following INPUT statement:

```
INPUT A$
```

When the user types a value in this data entry space and presses the Enter key, the IB runtime system strips trailing blanks (if any) from A$.

Here are some examples:

- Suppose you want to check if someone has entered a null (i.e., they pressed the Enter key without typing any data). The IF statement would look like this:

  ```
  IF A$ = "" THEN
  ```

- Suppose you want to compare the input value with a known string constant, such as “YES”. The INPUT statement would like this:

  ```
  IF A$ = "YES" THEN
  ```

- Suppose you want to compare the input value with a know string value that contains trailing blanks. There are two solutions to this problem. You could use the STRIPR function to strip the trailing blanks from the string value, or you could pad the input string with blanks (A$ = A$ + " ").

It’s possible to input multiple fields with the INPUT statement (see Chapter ??? for more information), in which case the blanks are stripped only from the final field in the INPUT statement (and then only if it’s a string field).

The PRINT statements strips trailing blanks

The IB runtime system strips trailing blanks from print/display string values, whether they are constant or variable. If you are printing multiple fields, blanks are stripped only from the final field (and then only if it’s a string). For example, suppose you have defined a string field named A$ with a length of 10 characters, and you use the following statements:

```
A$ = "HELLO"
PRINT A$
```

The IB runtime system strips trailing blanks (if any) from A$ and, in this case, prints 5 characters (HELLO).

If you want to print the defined number of characters, regardless of the value of the string field, add another field after the printed data, such as a null character. For example:

```
PRINT A$;"@00@"
```

In this case, when the PRINT statement is executed, all 10 characters of A$ are printed and the blanks are stripped from the last field only (the null).

Summary

Now you know the rules about writing programs in InternetBasic – most of the rules, that is. There are others, to be sure, and you'll learn them as we proceed through the other features in the IB language. In the next chapter, we talk about printing.