Introduction

Objectives

- To learn the role and representation of four basic data types: Integer, Boolean, Double, and String.
- To learn how to define variable names and to declare the type of data that will be associated with them.
- To learn how to define literals and constants in each of the four basic data types.
- To become familiar with the common arithmetic, Boolean, String, and relational operators and use them to construct assignment statements and arithmetic, logical, string, and relational expressions.
- To learn the rules of precedence for all the operators of the various data types.
Representing Data

The basic repository of data values is memory. In order to reference that memory, we use labels, called variable names (or more simply "variables"), that are assigned to the locations of memory where the data resides. Data values come in many forms, and each is represented differently in memory.

Numbers can be integers, real numbers, currency, dates, or times. Instead of numbers, the data values might be characters or sequences of characters, referred to as Strings. While Visual Basic can often determine the type of data from the way it is used in an expression, it is generally better for programmers to identify the kind of data explicitly. You can do so by providing type declarations that describe the kind of data that can be assigned to a variable: in other words, the kind of value that will be stored in the memory represented by the variable.

One of the ways to provide a type declaration is with a Dim statement. This statement identifies the name of a variable and the type of data that will be assigned to it. For example:

```
Dim toKms As Boolean
```

In this case, the variable toKms is to be assigned Boolean values: that is, True or False. Boolean is but one of a number of Visual Basic keywords whose purpose is to designate a data type. Four of the most common are:

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>to represent the truth values</td>
</tr>
<tr>
<td>Integer</td>
<td>to represent the positive and negative integers</td>
</tr>
<tr>
<td>Double</td>
<td>to represent real numbers</td>
</tr>
<tr>
<td>String</td>
<td>to represent sequences of characters as text</td>
</tr>
</tbody>
</table>

In this course, we will confine ourselves primarily to these basic types. The interested person can check out other types that are also available by accessing Visual Basic specification information provided on the Internet at the Microsoft Development Library. (Links to an external site.)

Because the size of memory allocated to the representation of each of these types is fixed, the magnitude and precision of the numbers that can be represented is limited. For example, the Integer data type is only capable of representing the integers in the following range:

```
-2,147,483,648 to +2,147,483,647
```
The Double data type can represent a larger range of numbers:

\[
\text{from about } -1.8 \times 10^{308} \text{ to about } +1.8 \times 10^{308}
\]

but only with at most 18 significant digits. Though this is more than sufficient for most scientific calculations, it is nevertheless finite.

In this course, your primary decision as a programmer about numerical data will be whether it will be fractional or too large to use variables declared as `Integer`. In that case, the `Double` data type should be adopted.

**Constructing Variable Names**

Variable names can be constructed from lower and upper case letters and digits. The underscore symbol, " \_ ", can also be used, but any other characters, including spaces, are not allowed. Furthermore, variable names cannot begin with a digit.

Because the number of characters is not a factor (up to 16,383 can be used), it is important to construct readable, descriptive name that captures the purpose of the variable. Such a name may be composed of several words with the space between replaced by underscores. Another common practice is to capitalize the first letter of each successive word after the first. The following example illustrates two possibilities for constructing:

Possible variable names from the phrase "seasonal adjusted amount":

```
seasonal_adjusted_amount
seasonalAdjustedAmount
```

As both examples show, it is also common practice not to capitalize the first letter of a variable name. The reason for not doing so is that another role for symbolic names is to label methods. You have already seen instances of such names in the previous units. By always capitalizing method names and never capitalizing variable names, it is easy to recognize at a glance whether an identifier in a program is the name of a method or a variable.

These rules for constructing variable names are examples of programming conventions—practices that are not rules. A program is not compromised if you fail to adopt or follow them. However, an important consideration in the design of any program is its "maintainability." This term refers to how easy it is for another person, not the original author of the program, to read and understand the logic of the program sufficiently to debug or make modifications to it. While it may take fewer keystrokes on your part to use the variable name "sam" instead of "seasonal_adjusted_amount", the latter is clearly more descriptive about its role in a program.
In your assignments, you will be expected to adopt the programming conventions suggested in this Study Guide and the textbook. View the marker of your assignment as being faced with the task of maintaining your program! The harder it is for the marker to "read" your source code, the less likely it is that you will receive full marks.

As described in the introduction, variables are defined using a "Dim" statement and this statement identifies the type of data that will be associated with each variable; that is, its data type. Like most programming languages, Visual Basic allows programmers to be "lazy" and use variables with different data types in an expression. Such expressions are called "mixed-mode" expressions. In these cases, Visual Basic will try to determine an appropriate data type for the value of the expression from the way the variables are used. This is not good practice, however, because sometimes the wrong inference is made. This can produce a working program that produces the wrong answer! This is a more serious form of programming error than a syntax error, which is what occurs if Visual Basic is unable to make any useful inference about type.

To help ensure that you do not mix modes, you should make sure that "Option Strict" is enabled. You can do this by clicking on "Options" in the "Tools" pull-down menu. Then click on the "arrowhead" to the left of "Projects and Solutions" and select "VB Defaults". Set the default project setting of "Option Strict" to "On". For more details, read page 71 of your textbook where Option Strict is described.

A single Dim statement can be used to declare several variables:

```
Dim num1 As Integer, num2 As Integer, pi As Double, status As Boolean
```

Alternatively, each variable can be declared in a succession of separate Dim statements:

```
Dim num1 As Integer
Dim num2 As Integer
Dim pi As Double
Dim status As Boolean
```

**Literals and Constants**

**Literals**

Four data types have been introduced—**Boolean**, **Integer**, **Double**, and **String**. Specific instances of each of these data types occur frequently in your programming. Such instances are called "literals." The representation of each type of literal has a specific syntax:
<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>DESCRIPTION</th>
<th>EXAMPLE(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>One of the truth values represented by True or False</td>
<td>True, False</td>
</tr>
<tr>
<td>Integer</td>
<td>A sequence of digits, without intervening commas or decimal points. Negative integers are preceded by a minus sign.</td>
<td>-123456</td>
</tr>
</tbody>
</table>

There are two ways:

1. *Decimal fraction*: a sequence of digits, followed by a decimal point, followed possibly by more digits. Negative numbers begin with a minus sign

   -12.345,

2. *Scientific notation*: A decimal fraction, expressing the significant digits, followed by the letter "E", followed by an integer representing the exponent. Negative numbers are preceded by a minus sign.

   -1234.5E-2

| Double     | A sequence of 0 or more characters, enclosed in double-quotes. | "abcd", "1a_% bcd 2#" |

**NOTE 1:** The Double literal -1234.5E-2 represents the value -1234.5 \times 10^{-2}.

**NOTE 2:** The String literal "1a_% bcd 2#" includes spaces. Space is a valid character in String literals. In any other type of literal it is either forbidden or ignored.

### Constants

A Visual Basic "constant" is a variable name used to represent specific value symbolically. A good example of using a symbol to represent a value is the use of the symbol "\( \pi \)" to represent the value 3.141592 \ldots to however many digits are needed. Since the Greek letters are not among the characters from which variable names can be constructed, a variable name such as "pi" can be used instead to represent this value wherever it is needed in the program provided we first initialize the variable to that value. We do so with a "Const" statement, which is normally placed at the beginning of the program. For example:

```vbnet
Const pi As Double = 3.1415926
```

Compare this with using the `Dim` statement, that was introduced earlier. The following is also allowed:

```vbnet
Dim pi As Double = 3.1415926
```
So, what’s the difference? The purpose of the **Dim** statement is to declare variables that are likely to change. In the **Dim** declaration statement, the assignment of an initial value is optional. The purpose of the **Const** statement is to assign a label to a constant value. The value of the label will not be changed by the program. Try the following:

1. Create a new project.
2. Double-click on the currently blank Form window. This will display source code consisting of one method, Form1_Load.
3. Following the Public Class declaration, type in the Const declaration for pi given above.
4. Within the body of the method, Form1_Load, type an assignment statement that assigns a different value to pi.

Your source listing should look something like:

```vbnet
Public Class Form1
    Const pi As Double = 3.1415926
End Class
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    pi = 1
End Sub
```

5. Press the F5 key on the keyboard to run your program. You will see a window open with a message that there were "build" errors.
6. Click on the "No" button to terminate execution.
7. Now look in the "Error List" window immediately below your source listing. You should observe:

![Error List](image)

This message tells you that you cannot assign a value to a variable name that has been declared to be a constant. If you repeat the same steps replacing **Const** by **Dim**, no error will be detected because none has occurred.

The ability to declare constants provides an opportunity to introduce an important programming convention:

Avoid using the actual values of constants, particularly well-known ones as literals in your program. Instead, assign them symbolic names using a **Const** declaration statement.
The reason for this practice is twofold:

1. The choice of a meaningful name will better clarify the significance of the constant you are using.
2. Any decision to change the value of the constant requires changing the Const statement only. This fact will save you the effort of finding all the occurrences of the value as a literal that need to be changed throughout the program.

Evaluating Numeric Expressions

Associated with each of the data types is a set of operators and functions, and many of these will be familiar to you. Most familiar should be the elementary arithmetic operations for adding, subtracting, multiplying, and dividing. The VB symbols for these operators are, not surprisingly, "+", "-", "+", and "/" respectively. Since these are all binary operators, they require two operands in order to express a calculation. More complex expressions can be created when a sequence of operand and operators are used to define a calculation. For example: \( 1 + 2 \times 4 / 3 - 5 \).

What is not so clear is how this expression will be evaluated in Visual Basic. There are different results possible depending on the order in which the operators are applied to their respective operands:

1. If we evaluate the expression from left to right, then we obtain \(-1\).
2. If we evaluate the expression from right to left, then we obtain \(-3\).
3. If we evaluate the multiplication first, then division, then addition and finally subtraction we obtain \(-2.33\) . .

There are other possibilities, but the point is that rules are required that dictate how to evaluate arithmetic expressions.

One simple solution can be provided by introducing parentheses into the expression along with the rule:

**Parentheses Rule:** Whatever is in parentheses is evaluated first.

By introducing parentheses in different ways we can construct expressions that evaluate to the different results:

\[
((1 + 2) \times 4) / 3 - 5 \text{ evaluates to } -1 \\
1 + (2 \times (4 / (3 - 5))) \text{ evaluates to } -3 \\
(1 + ((2 \times 4) / 3)) - 5 \text{ evaluates to } -2.33 . .
\]

While this rule and a liberal use of parentheses ensures that there is only one way to evaluate the expression, the introduction of the parentheses makes the expression less readable.
As an alternative, we can provide rules that give priority to certain operations over others. These rules, called the Rules of Precedence, are:

1. Multiplication and division are performed before addition or subtraction and from left to right.
2. Addition and subtraction are then performed, again from left to right.

According to these rules, the "correct" value for the example expression given is \(-2.33\).

Finally, we may wish to perform addition before multiplication in some expressions, and in such cases, parentheses along with the parentheses rule can be used to ensure that we can write an expression that will be evaluated in the desired order. For example

\[(1 + 2) \times 3 / 4 - 5\] evaluates to \(-2.75\)

You may have noticed that all the values in the expression are integer, yet when the precedence rules are applied the result is the Double data type. The reason is that during the evaluation, the result was not an integer when division was performed. From that point on, the remaining evaluations were performed as if the remaining operands were of type Double. When the type of a value is changed in order to complete the evaluation of an expression, we say a "cast" has occurred. Casting is convenient, but it may have unintended results. So it is best to avoid writing expressions where casting is required. Remember that earlier in this unit, you were advised to set "Option Strict" to "On" at the beginning of every program. If you have done so, then any statements where casting is required will be detected and identified for you.

In the example given, we can avoid the cast occurring if we simply express all the values as Doubles rather than as Integers:

\[(1. + 2.) \times 3. / 4. - 5.\]

In this example it makes no difference which way we represented the integers if in either case we expected the result to be \(-2.75\). But suppose we wanted to perform "integer division": that is, we want to obtain a quotient and discard the remainder. In that case, we need an "integer division operator" rather than "/", and Visual Basic provides one: "\".

Then, to perform integer division in our original example, we write the expression as:

\[(1 + 2) \times 3 \div 4 - 5\]

Applying the rules of precedence, the value is \(-3\) since the quotient obtained from dividing 9 by 4 is 2.

One further operator that can be applied to numeric data is the power operator: ":^:".

The VB expression:

\[a ^ b\]

means "take a to the power b."
The power operator requires another precedence rule. The following list gives this precedence rule and summarizes all the precedence rules for arithmetic operators:

**Arithmetic Precedence Rules**

1. Evaluate all expressions inside parentheses.
2. Evaluate all expressions that employ the power operator from right to left.
3. Evaluate all multiplications and divisions from left to right.
4. Evaluate all additions and subtractions from left to right.

Note that, unlike the other four arithmetic operators, when the expression contains more than one power operator, it is evaluated from right to left (and not left to right)—for example:

\[
2^2^3 = 256 = 2^{(2^3)},
\]

\[\text{not } (2^2)^3 = 64\]

When in doubt, use parentheses to ensure that the expression is evaluated in the order you expect.

**Calculations on Other Data Types**

The arithmetic operators are likely to be the most familiar to you. However, calculations can also be performed with Boolean values and Strings.

**Boolean Calculations**

The Boolean operators, also called the logical operators, are And, Or, and Not. Since these operators have operands whose values are either True or False, the values these operators produce are also either True or False.

The values obtained from applying these operators are as follows:

- a And b evaluates to True if and only if both a and b have the value True.
- a Or b evaluates to True if either a or b is True. This includes the case when both a and b are True; that is, True Or True evaluates to True.
- Not a evaluates to True if a is False and evaluates to False if a is True.

Like the arithmetic operators, precedence rules are required to evaluate expressions that contain more than one operator. The precedence rules for Boolean expressions are as follows:
Boolean Precedence Rules

1. Evaluate all expressions inside parentheses.
2. Evaluate all "Not" expressions right to left.
3. Evaluate all "And" expressions. Since "And" is commutative, the order of evaluation can be either left to right or right to left.
4. Evaluate all "Or" expressions. Again, since "Or" is commutative, expressions with multiple "Or" operators can be evaluated in either direction and will produce the same result.

String Calculations

Strings have but one operator, called "concatenation," represented by the symbol "&." The effect of concatenation is to extend the first (i.e., left) operand by attaching onto its right end the string specified by the second (i.e., right) operand. The following VB source code illustrates the operation:

```vbnet
Const first_name As String = "Simon"
Const last_name As String = "Fraser"
Const institution As String = "University"

Dim institution_name As String
institution_name = first_name & last_name & institution
```

The value of institution_name is "SimonFraserUniveristy"

Note that if spaces are desired between the words in the institution_name then the assignment statement needs to be revised by inserting a blank character, " ", between each word:

```vbnet
institution_name = "first_name" & " " & second_name & " " & institution
```

Concatenation is an associative operation, and so expressions containing multiple concatenation operators can be evaluated either left to right or right to left.

Relational Operators

There is another group of operators with which you will likely be familiar if you have had to compare two numeric values. These operators are called "relational operators" and are defined as follows:
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equals</td>
</tr>
<tr>
<td>=</td>
<td>equals</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equals</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equals</td>
</tr>
</tbody>
</table>

From a programming perspective, these operators are different from the operators described previously because relational expressions take different types of operands but evaluate to Boolean values; that is, True or False. So, for example:

- 249.5 < 303.25 evaluates to True
- 150 >= 250 evaluates to False.

The relational operators can also be used to compare Strings. In this case, alphabetical ordering is used to determine whether a relational expression should evaluate to True or False. So

- "CMPT 110" < "MACM 101" evaluates to True because the String "CMPT 110" comes alphabetically before "MACM 101".

You must take care, however, when comparing Strings that use characters other than the letters of the alphabet or that use both lower and upper case. In the computer everything is represented by binary numbers (i.e., base 2). This includes all the characters, including the digits, upper and lower case characters, and all the special characters. Every character is given a unique binary code word, called its ANSI value. It is the code words that are compared when two Strings are compared in a relational expression. For example,

- "cmpt 110" < "Macm 101" evaluates to False because the code word for the letter "c" is the binary number 01100011 (or 99 in decimal), while the code word for the letter "M" is 01001101 (or 77 in decimal). Since 77 < 109, the character "M" comes before the character "c."

A complete listing of the ANSI code words for all the characters is at the following website: [https://www.medcalc.org/manual/ansi_character_set.php](https://www.medcalc.org/manual/ansi_character_set.php) and is useful in determining the "alphabetical" ordering of all the characters.

Another common mistake is to forget that "blank" or "space" is a valid character (its ANSI codeword is 00110000). In any string where it occurs, it will be compared with the character in the same position of another string. So

- "CMPT 110" = CMPT110 " evaluates to False
because the fifth character in "CMPT 110" is " " while the fifth character in "CMPT110 " is "1".

Finally, any two operands compared with a relational operator defines an expression that has a value, either True or False. Therefore its value can be assigned to a variable of type Boolean. As an example:

```python
test_val = x >= 25
```

the Boolean variable test_val will be assigned the value True if the value of the numeric variable x is greater than or equal to 25.