Introduction to VISUAL BASIC

2016 edition

Copyright © HKTA Tang Hin Memorial Secondary School 2017
# Table of Contents

Chapter 1  Select...Case Statement  
  1.1  Syntax of Select...Case statement  
  1.2  Examples  
  1.3  Official syntax  
  Exercise 1  

Chapter 2  For...Next Statement  
  2.1  Step keyword  
  2.2  Using the debugger  
  2.3  Exit For and Continue For  
  2.4  Nesting For...Next statements  
  2.5  Miscellaneous examples  
  2.6  Official syntax  
  Exercise 2  

Chapter 3  More on Variables  
  3.1  Decimal data type  
  3.2  Data types of literals  
  3.3  Scope of variables  
  3.4  Best practices with variables  

Glossary  


Chapter 1  Select...Case Statement

In the past, we have learned how to do decision making with If...Then...Else statements. Usually, we make decision based on the value of a variable, such as the marks of a test, or a randomly drawn number. Sometimes these decisions can be written more efficiently with Select...Case statement.

Example: lucky draw

With If...Then...Else statement, the program looks like this:

```vbnet
Randomize()

Console.WriteLine("Lucky Draw")
Console.WriteLine("==========")

Dim draw As Integer = Int(Rnd() * 100) + 1
If draw <= 50 Then
    Console.WriteLine("No Prize")
ElseIf draw <= 75 Then
    Console.WriteLine("Small Prize")
ElseIf draw <= 90 Then
    Console.WriteLine("Medium Prize")
ElseIf draw <= 99 Then
    Console.WriteLine("Big Prize!")
Else
    Console.WriteLine("JUMBO Prize!!!")
End If

Console.ReadLine()
```

Lucky Draw
==========
JUMBO Prize!!!
Now the same program is written using `Select ... Case` statement. Which program is more readable?

```vbnet
Randomize()

Console.WriteLine("Lucky Draw")
Console.WriteLine("==========")

Dim draw As Integer = Int(Rnd() * 100) + 1
Select Case draw
    Case 1 To 50
        Console.WriteLine("No Prize")
    Case 51 To 75
        Console.WriteLine("Small Prize")
    Case 76 To 90
        Console.WriteLine("Medium Prize")
    Case 91 To 99
        Console.WriteLine("Big Prize!")
    Case Else
        Console.WriteLine("JUMBO Prize!!!")
End Select
```
1.1 Syntax of Select...Case statement

The syntax of a Select...Case statement is shown here:

```
Select Case [testexpression]
    Case [expressionlist1]
        statements1
    Case [expressionlist2]
        statements2
    Case Else
        elsestatements
End Select
```
You can see the flowchart for the execution of the `Select ... Case` statement above. Only the statements below the first matched expression list is executed. For example, if `testexpression` matches `expressionlist1`, then `statements1` is executed. All subsequent expression lists, from `expressionlist2`, are skipped because of the match.

If `testexpression` does not match `expressionlist1`, then it is matched against `expressionlist2`. If `testexpression` again does not match `expressionlist2`, then it is matched against `expressionlist3`, etc.

If none of the expression lists are matched, then the statements below `Case Else` (if any) are executed. Like the `Else` keyword in `If ... Then ... Else` statements, `Case Else` can be omitted in `Select ... Case` statements.

It is possible to have nothing in `statements1`, `statements2`, etc. In this case, a comment should be added in place of the statement to make the program more readable.

**Test expression**

`testexpression` must be an expression that evaluates to an elementary data type such as `Integer`, `Single`, `Double`, `Boolean` and `String`.

You can do something like `Select Case mass / (height * height)`.
Expression lists

Each expression list contains one or more expression clauses, and each clause can take one of the following forms:

<table>
<thead>
<tr>
<th>Expression Clause</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Is equal to the expression</td>
<td>100</td>
</tr>
<tr>
<td>expression1 To expression2</td>
<td>Between expression1 and expression2 (inclusive)</td>
<td>15 To 19 &quot;A&quot; To &quot;Z&quot;</td>
</tr>
<tr>
<td>Is [=/&lt;/&lt;=/&gt;/&gt;=/&lt;&gt;] expression</td>
<td>Satisfies the given condition</td>
<td>Is &gt; 200 Is &lt;= &quot;ba&quot;</td>
</tr>
</tbody>
</table>

Multiple clauses are separated by commas. Here are some examples:

```
Case "angel"
Case 0 To 100
Case Is > 0
Case "A" To "Z", "a" To "z"
Case 1 To 4, 7 To 9, 11, 13, Is > maxNumber
```

Note: all items in the expression list are converted to the same type as `testexpression` before comparison is made. This may produce unexpected results, for example:

```
Dim x As Integer = 4
Select Case x
    Case Is <= 3.5
        ' The following statement gets executed!
        Console.WriteLine("<= 3.5")
End Select
```
1.2 Examples

Simple example with strings

Console.Write("Enter the grade (A-F): ")
Dim grade As String = UCase(Console.ReadLine())

Select Case grade
    Case "A", "B", "C", "D", "E"
        Console.WriteLine("Pass")
    Case "F"
        Console.WriteLine("Fail")
    Case Else
        Console.WriteLine("Not a valid grade")
End Select
Console.ReadLine()
Example dealing with floating-point numbers

```csharp
Console.Write("Enter the marks: ")
Dim marks As Double = Console.ReadLine()

Dim grade As String
Select Case marks
    Case 80 To 100
        grade = "A"
    Case 70 To 80
        grade = "B"
    Case 50 To 70
        grade = "C"
    Case 0 To 50
        grade = "D"
    Case Else
        grade = "Error"
End Select

If grade = "Error" Then
    Console.WriteLine("The mark must be between 0 and 100.")
Else
    Console.WriteLine("Your grade is {0}.", grade)
End If
Console.ReadLine()
```

Enter the marks: 79.5
Your grade is B.

If the statement Case 70 To 80 is written as Case 70 To 79 or Case 70 To 79.9, the program will not run correctly. However, it is perfectly OK to write Case 70 To 81. Why?
Class work

Complete the programs below by completing the blanks.

**Program 1**: throwing a die. The player gets a bonus by throwing a six.

```vbnet
Randomize()

Dim x As Integer = Int(Rnd() * 6) + 1
Select Case x
  ______________________
    Console.WriteLine("Normal")
  ______________________
    Console.WriteLine("A six!!!")
End Select
```
**Program 2**: BMI calculator. After the BMI is calculated, classify the results with the table below:

<table>
<thead>
<tr>
<th>Body mass index (BMI)</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 18.5</td>
<td>Underweight</td>
</tr>
<tr>
<td>≥ 18.5 and &lt; 24</td>
<td>Average</td>
</tr>
<tr>
<td>≥ 24 and &lt; 28</td>
<td>Overweight</td>
</tr>
<tr>
<td>≥ 28</td>
<td>Obese</td>
</tr>
</tbody>
</table>

```vbnet
Console.Write("Enter the mass (in kg): ")
Dim mass As Double = Console.ReadLine()

Console.Write("Enter the height (in cm): ")
Dim height As Double = Console.ReadLine() * 0.01

Dim BMI As Double = mass / (height * height)
Dim classification As String

classification = "Underweight"
classification = "Average"
classification = "Overweight"
classification = "Obese"

Console.WriteLine("BMI: {0}, Result: {1}", BMI, classification)
```
1.3 Official syntax

Here is the official syntax of the Select...Case statement. The parts inside [] are optional.

```
Select [ Case ] testexpression
    [ Case expressionlist
        [ statements ] ]
    [ Case Else
        [ elsestatements ] ]
End Select
```

When you enter code into the Visual Studio, it changes Select into Select Case if the word Case is not already there. Anyway, the program can compile and run correctly even without these additional Case.

Also, in an expressionlist, the keyword Is before the comparison operators is optional.
Exercise 1

1. Write a program that the user will enter the amount of pocket money he/she spends per week. Then the program will output a message according to the table below. You must use Select ... Case statement in your program to do the task.

<table>
<thead>
<tr>
<th>Pocket Money ($)</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>Negative! Are you serious?</td>
</tr>
<tr>
<td>0</td>
<td>You didn’t spend any money. Are you lying?</td>
</tr>
<tr>
<td>&gt; 0 and &lt; 100</td>
<td>Below average. What a good student!</td>
</tr>
<tr>
<td>≥ 100 and &lt; 200</td>
<td>Average.</td>
</tr>
<tr>
<td>≥ 200</td>
<td>Above average. Consider to spend less!</td>
</tr>
</tbody>
</table>

Enter the pocket money you spend per week: -3
Negative! Are you serious?

2. Write a program that the user enters a string. By using Select ... Case statement, output the following message according to the conditions below. (Hint: use a string function to extract the first character of the string.)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starts with an upper case letter</td>
<td>The string starts with an upper case letter.</td>
</tr>
<tr>
<td>Starts with a lower case letter</td>
<td>The string starts with a lower case letter.</td>
</tr>
<tr>
<td>Starts with a digit</td>
<td>The string starts with a digit.</td>
</tr>
<tr>
<td>Others</td>
<td>The string starts with something strange.</td>
</tr>
</tbody>
</table>

Enter a string: Visual Basic is easy!!!
The string starts with an upper case letter.
3. Write a program to find the n-th triangle number. To make Select...Case statement necessary, follow the sample output below:

```
Enter n: 101
The 101st triangle number is 5151.
```

Hint: you may want to write the code generating the letters “st”, “nd”, “rd” or “th” in a function.
Chapter 2  For...Next Statement

For...Next statement is used to repeat a block of instruction a specified number of times. A typical For...Next statement looks like the one below:

```
For counter [As type] = start To end
    statements
Next
```

Note: The statement `Next` can also be written as `Next counter`.

In the loop above, we say that the `counter` counts from `start` to `end`, taking a different value in each iteration. Or more precisely, it does the following:

```
counter = start
Do While counter <= end
    statements
    counter += 1
Loop
```

Besides the scope of variables, there is actually some subtle differences between the two code snippets above. Check the last part of this chapter for details.
A simple example

```
For i As Integer = 1 To 10
    Console.WriteLine(i * i)
Next
Console.ReadLine()
```

In the program above, we declare variable `i` and use it as the counter. The value of `i` counts from 1 to 10, and `Console.WriteLine(i * i)` is executed for each value of `i`. Therefore, the value of the first 10 square numbers (`i * i`) are printed out.

You can declare the variable counter before the `For...Next` statement, but this is not recommended. In most cases, Visual Basic creates the counter variable and limits its scope to the `For...Next` statement.

How a `For...Next` statement ends execution

The next example investigates the value of the counter variable (`i`) during and after the `For...Next` statement.

In the example, we need to declare `i` before the loop. If we write `For i As Integer = ...` instead, it is impossible to read the value of `i` after the loop.
Putting `For i As Integer = ...` next to `Dim i As Integer` results in a compile error. Details will be discussed in the next chapter.

During the loop, the value of `i` goes from 3 to 7, increasing by one in each iteration. The loop ends when the value of `i` is increased from 7 to 8.

As described before, the program works this way:
Another simple example

This example calculates the value of the first 10 triangle numbers.

```vbnet
For i As Integer = 1 To 10
    Dim t As Integer = i * (i + 1) \ 2
    Console.WriteLine("i = {0}, t = {1}", i, t)
Next
Console.ReadLine()
```

```
i = 1, t = 1
i = 2, t = 3
i = 3, t = 6
i = 4, t = 10
i = 5, t = 15
i = 6, t = 21
i = 7, t = 28
i = 8, t = 36
i = 9, t = 45
i = 10, t = 55
```
2.1 Step keyword

In Visual Basic, it is possible to set the counter to increase or decrease at any value after every iteration. This is done using the Step keyword followed by an expression.

For example, the following statement counts from 10 down to 1.

```vbnet
For i As Integer = 10 To 1 Step -1
    statements
Next
```

And the following iterates every odd number from 1 to 100:

```vbnet
For i As Integer = 1 To 100 Step 2
    statements
Next
```

Class work

Write down the For statement if you want the counter variable `i` to be the values in the lists below. The first question is done for you as an example:

<table>
<thead>
<tr>
<th>Values of variable <code>i</code></th>
<th>For...Next statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, ..., 100</td>
<td>For <code>i</code> As Integer = 1 To 100</td>
</tr>
<tr>
<td>-3, -2, -1, 0, 1, 2, 3</td>
<td></td>
</tr>
<tr>
<td>7, 6, 5, 4, 3, 2</td>
<td></td>
</tr>
<tr>
<td>101, 103, 105, ..., 199</td>
<td></td>
</tr>
<tr>
<td>100, 98, 96, 94, ..., 2</td>
<td></td>
</tr>
<tr>
<td>1, 4, 7, 10, ..., 100</td>
<td></td>
</tr>
</tbody>
</table>

- If the step is positive or zero, the condition to execute the loop is `counter <= end`. Otherwise, the condition is `counter >= end` instead.
Appendix: increase/decrease by a decimal

You can use a decimal with the `Step` keyword. However, if `Single` or `Double` is used, unexpected results may happen due to round-off error.

```vbnet
For i As Single = 7 To 10 Step 0.6  ' This will run incorrectly.
    Console.WriteLine("i = " & i)
Next
```

Discussion: why is there no iteration with `i = 10`? If `Single` is changed to `Double`, will there be any difference? And how about changing 0.6 to another number, such as 0.3?

Avoid using `Single` and `Double` as a counter of For...Next statement.
2.2 Using the debugger

To understand the use of `For ... Next` statements, we can use the **Locals** windows of Visual Studio. The Locals window inspects the value of all local variables.

Instead of running the whole program, press **F11** (Step Into) to run the program step by step. You can also use **F9** to create breakpoints so execution stops at a given point.

The screen capture below shows the progress of a running program.

![The Debugger in Visual Studio](image)

**Figure 1.** The Debugger in Visual Studio
2.3 Exit For and Continue For

Exit For means to exit the For...Next statement immediately. Control is transferred to the statement after the loop.

Continue For means to bypass everything in the current iteration. The counter variable is incremented, and control goes to the next iteration if the loop condition is satisfied.

Example: ascending order

```vbnet
Console.WriteLine("Enter 5 numbers in ascending order.")

Console.Write("Number 1: ")
Dim currentTerm As Double = Console.ReadLine()

Dim isAscending As Boolean = True
For i As Integer = 2 To 5
    Console.Write("Number {0}: ", i)
    Dim nextTerm As Double = Console.ReadLine()
    If currentTerm > nextTerm Then
        isAscending = False
        Exit For
    End If
    currentTerm = nextTerm
Next

Console.WriteLine()

If isAscending Then
    Console.WriteLine("Correct!")
Else
    Console.WriteLine("Incorrect!")
End If

Console.ReadLine()
```
Enter 5 numbers in ascending order.
Number 1: 1
Number 2: 3
Number 3: 3
Number 4: 4
Number 5: 6
Correct!

Enter 5 numbers in ascending order.
Number 1: 7
Number 2: 9
Number 3: 5
Incorrect!
Example: skipping numbers with digit 4

' Similar to "Instr(n, digit) > 0", but more efficient.
Function HasDigit(n As Integer, digit as Integer) As Boolean
    n = Math.Abs(n) ' Remove minus sign.
    Do While n <> 0
        If n Mod 10 = digit Then
            Return True
        End If
        n = n \ 10
    Loop
    Return False
End Function

Sub Main()
    For i As Integer = 1 To 100
        If HasDigit(i, 4) Then
            Continue For
        End If

        Console.Write(i & " ")
        If i Mod 10 = 0 Then
            Console.WriteLine()
        End If
    Next
    Console.ReadLine()
End Sub
2.4 Nesting For...Next statements

Like all other statement blocks, For...Next statement can be nested. Of course, a different counter variable is needed for each layer of For...Next statement.

The next example shows how nesting works. You should note how `Console.Write` and `Console.WriteLine` produce a table here.

```csharp
For i As Integer = 1 To 5
    For j As Integer = 1 To 7
        Console.Write("{0}-{1} ", i, j)
    Next
    Console.WriteLine()
Next
Console.ReadLine()
```

```
1-1 1-2 1-3 1-4 1-5 1-6 1-7
2-1 2-2 2-3 2-4 2-5 2-6 2-7
3-1 3-2 3-3 3-4 3-5 3-6 3-7
4-1 4-2 4-3 4-4 4-5 4-6 4-7
5-1 5-2 5-3 5-4 5-5 5-6 5-7
```

The value of the counter in the outer loop can be used in the For...Next statement in the inner loop. See the example below:

```csharp
For i As Integer = 1 To 6
    For j As Integer = i To 6
        Console.Write(j)
    Next
    Console.WriteLine()
Next
Console.ReadLine()
```
Here is another example:

```vbnet
Console.Write("Enter the number of rows: ")
Dim numOfRows As Integer = Val(Console.ReadLine())
For i As Integer = 1 To numOfRows
    Dim numOfSpaces As Integer = numOfRows - i
    For j As Integer = 1 To numOfSpaces
        Console.Write(" ")
    Next
    Dim numOfStars As Integer = i * 2 - 1
    For j As Integer = 1 To numOfStars
        Console.Write("*")
    Next
    Console.WriteLine()
Next
Console.ReadLine()
```

```
Enter the number of rows: 9
   *
  ***
 *****
*******
********
*********
**********
***********
************
*************
```

---

**Introduction to Visual Basic (Part 2)**
Class work

Write a program that produces the output shown below. The shaded texts are user inputs.

```
Enter the number of rows: 4
#
##
###
####
```

```csharp
Console.WriteLine("Enter the number of rows: ")
Dim numOfRows As Integer = Console.ReadLine()

Console.ReadLine()
```
2.5 Miscellaneous examples

Finding the sum of squares

```vbnet
Console.Write("Enter the value of n: ")
Dim n As Integer = Console.ReadLine()
Dim sum As Integer = 0
For i As Integer = 1 To n
    sum += i * i
Next
Console.WriteLine("The sum of the first {0} squares is {1}.", n, sum)
Console.ReadLine()
```

Enter the value of n: 10
The sum of the first 10 squares is 365.

Reverse the content of a string

```vbnet
Console.Write("Enter a string: ")
Dim str As String = Console.ReadLine()
Dim reversed As String ="
For i As Integer = Len(str) To 1 Step -1
    reversed &= Mid(str, i, 1)
Next
Console.WriteLine("The reversed string is "{0}"., reversed)
Console.ReadLine()
```

Enter a string: Reversing a string
The reversed string is "gnirts a gnisreveR".
Clapping game "Sevens"

Answers of Game "Sevens"
[ * = Clap ]

1 2 3 4 5 6 * 8 9 10
11 12 13 * 15 16 * 18 19 20
* 22 23 24 25 26 * * 29 30
31 32 33 34 * 36 * 38 39 40
41 * 43 44 45 46 * 48 * 50
51 52 53 54 55 * * 58 59 60
61 62 * 64 65 66 * 68 69 *
* * * * * * * * 80
81 82 83 * 85 86 * 88 89 90
* 92 93 94 95 96 * * 99 100

As discussed before in this chapter, \texttt{Instr(i, "7")} > 0 is not efficient. A more efficient implementation is given in an earlier example.
Checking if a string represents an integer

' Returns if the string represents an integer.
Function IsInteger(str As String) As Boolean
    If str = "" Or str = "+" Or str = "-" Then
        Return False
    End If

    Dim ch = Left(str, 1)
    If (ch < "0" Or ch > "9") And ch <> "+" And ch <> "-" Then
        Return False
    End If

    For index As Integer = 2 To Len(str)
        ch = Mid(str, index, 1)
        If ch < "0" Or ch > "9" Then
            Return False
        End If
    Next

    Return True
End Function

Sub Main()
    Console.Write("Enter a string: ")
    Dim str As String = Console.ReadLine()

    If IsInteger(str) Then
        Console.WriteLine("It is an integer.")
    Else
        Console.WriteLine("It is not an integer.")
    End If

    Console.ReadLine()
End Sub

Enter a string: -653428
It is an integer.
To make a program more readable, divide the program into procedures (i.e. `Sub` and `Function`).


Enter a string: 36a2
It is not an integer.

Enter a string: +
It is not an integer.
Prime numbers

To know whether \( n \) is prime, we need to check whether \( x \) divides \( n \) for any \( x \) in \( 2 \leq x \leq \sqrt{n} \). Special considerations should be given to 0, 1, 2 and negative numbers.

If you need to list larger primes, such as from 1 to 10000000, the program above is not efficient enough. For a more efficient approach, see Sieve of Eratosthenes.

```vbnet
Function isPrime(n As Integer) As Boolean
    n = Math.Abs(n)
    If n < 2 Then
        Return False ' 0 and 1 are not primes.
    End If
    For i As Integer = 2 To CInt(Int(Math.Sqrt(n)))
        If n Mod i = 0 Then
            Return False
        End If
    Next i
    Return True
End Function

Sub Main
    Dim upperLimit As Integer = 100
    Console.WriteLine("The prime numbers up to {0} are:", upperLimit)
    For i As Integer = 2 To upperLimit
        If isPrime(i) Then
            Console.Write("{0} ", i)
        End If
    Next
    Console.ReadLine()
End Sub
```
Fibonacci series

```vbnet
Console.WriteLine("Fibonacci Series")
Console.WriteLine("================")

Console.Write("How many terms do you want? ")
Dim numOfTerms As Integer = Console.ReadLine()

If numOfTerms <= 0 Then
    Console.WriteLine("Please enter a positive number.")
Else
    Console.WriteLine("The first {0} terms of the Fibonacci series are:", numOfTerms)
    ' The first term
    Console.Write("1")
    ' The second term onwards
    If numOfTerms >= 2 Then
        Dim a As Integer = 0  ' The extended zeroth term
        Dim b As Integer = 1  ' The first term
        For i As Integer = 2 To numOfTerms
            ' Move "a" and "b" forward by one term.
            Dim c As Integer = a + b
            a = b
            b = c
            Console.Write("", {0}", b)
        Next
    End If
End If

Console.ReadLine()
```

Fibonacci Series
================

How many terms do you want? 12

The first 12 terms of the Fibonacci series are:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
2.6 Official syntax

Here is the official syntax of the For...Next statement. The parts inside [] are optional.

```
For counter [ As datatype ] = start To end [ Step step ]
    [ statements ]
    [ Continue For ]
    [ statements ]
    [ Exit For ]
    [ statements ]
Next [ counter ]
```

When a For...Next loop starts, the values of start, end and step are evaluated and used throughout the loop. These computed values do not change even if the values of the underlying variables of start, end and step change during the loop.

The value of the counter is set to be start at the beginning of the loop. If the counter is not already defined, VB.NET creates a variable that is scoped to the For...Next block. Details of scoping will be discussed in the next chapter.

If the data type of the counter is stated, VB.NET creates a variable even if it is already defined outside of the procedure. However, if the variable is defined inside the procedure, a compile error occurs.

The value of step determines the condition to execute the loop. If it is positive or zero, the loop executes if counter <= end, otherwise it executes if counter >= end. If the Step keyword is absent, the value of step is 1.

Exercise 2

1. Write a program that
   (a) prints out the first 10 triangle numbers.
   (b) prints out the first 100 triangle numbers, 10 in a row.

2. Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number, and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

3. Write a program that prints the numbers from 11 to 100, skipping all multiples of 2, 3, 5 and 7. (These numbers are prime numbers from 11 to 100.)

4. Write a program that finds the sum of the first \( n \) triangle numbers. The value of \( n \) should be inputted by the user.

   Enter the value of \( n \): 10
   The sum of the first 10 triangle numbers is 220.

5. Write a program that calculates the sum and average of a few numbers. The exact count of numbers should be determined by the user.

   How many numbers do you want to enter? 3
   Enter number 1: 23
   Enter number 2: 5.5
   Enter number 3: -3
   The sum is 25.5, and the average is 8.5.
6. Write a program that prints the pattern in the sample output. The number of rows (from 1 to 9) should be chosen by the user.

```
Enter the number of rows: 6
1
121
12321
1234321
123454321
12345654321
```

7. Write a program that prints an addition table (or multiplication table if you like challenges). (Note: add spaces to make numbers align nicely.)

```
Enter the number of columns: 10
Enter the number of rows: 4
+ | 1 2 3 4 5 6 7 8 9 10
----+-----------------------------------------
1 | 2 3 4 5 6 7 8 9 10 11
2 | 3 4 5 6 7 8 9 10 11 12
3 | 4 5 6 7 8 9 10 11 12 13
4 | 5 6 7 8 9 10 11 12 13 14
```

(Hint: See https://docs.microsoft.com/en-us/dotnet/standard/base-types/composite-formatting#alignment-component for extra tips using Console.WriteLine. This is especially useful for multiplication tables.)

8. Write a program to find all 3-digit numbers that have the following property:

The number is equal to the sum of the cubes of its digits.

e.g. $371 = 3^3 + 7^3 + 1^3$. You may print out the result in any format. (Hint: There are totally 4 answers, including 371.)
9. Write a program that
   (a) list out the positive factors of a number.
       (e.g. the positive factors of 6 are 1, 2, 3, and 6.)
   (b) finds out all “perfect numbers” from 1 to 1000. A “perfect number” is a number equal to the sum of all its positive factors excluding itself. Print the result in the following way:

   
   6 = 1 + 2 + 3
   28 = 1 + 2 + 4 + 7 + 14

10. Write a program that asks the player to do a set of (e.g. 10) addition problems. Output the score (e.g. “You have answered 8 out of 10 questions correctly.”) after all the questions are attempted.

11. Write a program that output a christmas tree, formed by two isosceles triangles and a rectangle connected together. The size of both triangles and the rectangle are given by the user. However, since the tree must be symmetric, the width of the stem is rounded down to the nearest odd number. (Note: the tree must be symmetric even if the sizes are in wrong proportions.)

   Enter the height of the top triangle: 3
   Enter the height of the bottom triangle: 4
   Enter the width of the stem: 1
   Enter the height of the stem: 2

   *
   ***
   *****
   *
   ***
   *****
   ********
   |   |
12. Write a program that converts a fraction to a terminating decimal or a recurring decimal.

Enter the numerator: 1423
Enter the denominator: 70
. . .
20.3285714

Enter the numerator: 3
Enter the denominator: -20
-0.15

Enter the numerator: 1
Enter the denominator: 98
. . .
0.0102040816326530612244897959183673469387755
Chapter 3  More on Variables

In this chapter we learn more about variables in VB.NET. The contents in this chapter are NOT required in examinations. Here is a list of topics in this chapter:

- **Decimal data type**
- **Data types of literals**
- **Scope of variables**
3.1 Decimal data type

Decimal is a data type in VB.NET that stores decimals in denary notation. Therefore, decimals such as 0.1 can be stored with Decimal type in its exact value. Decimal supports 28-29 significant digits and can represent values up to $7.9228 \times 10^{28}$.

In contrast, Single and Double store decimals in binary notation. Decimals such as 0.1 is a recurring decimal in binary notation, and hence cannot be stored with its exact value in Single and Double.

Values in Decimal type can be rounded off in the same way as in Mathematics. However, unexpected results may occur when you round off a value that already contains round-off errors. Here is an example:

```vbnet
Dim x As Decimal = 1.2345D ' D = Decimal

' Rounds off "x" to 3 decimal places.
Dim y As Decimal = Math.Round(x, 3, MidpointRounding.AwayFromZero)
```

Always add the letter D to denote a Decimal value. Otherwise it will be interpreted as a Double and rounding errors may occur.

Decimal is the slowest numeric data type in VB.NET. So use it only if necessary.

3.2 Data types of literals

Literals are values that you write in the source code, e.g. 123, True, "hello", etc. Like variables, literals have a data type. You can specify the data type of some literals with an ending character:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Character</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>I</td>
<td>123I</td>
</tr>
<tr>
<td>Long</td>
<td>L</td>
<td>123L</td>
</tr>
<tr>
<td>Single</td>
<td>F</td>
<td>12.34F</td>
</tr>
<tr>
<td>Double</td>
<td>R</td>
<td>12.34R</td>
</tr>
<tr>
<td>Decimal</td>
<td>D</td>
<td>12.34D</td>
</tr>
</tbody>
</table>

Note: Long is a 64-bit integer, while Integer is a 32-bit integer. Long can be used to store values larger than Integer.

There are no ending characters for Boolean and String. If there is no ending character, the data type is Integer, Long, Double, Boolean or String, depending on the content of the literal.

Sometimes you must write down the data type of the literal for the program to compile or run correctly. For example:

```vbnet
dim a as long = 123456 * 123456 ' Compile error.
dim b as long = 123456L * 123456L ' OK.
```

In the example above, we have $123456^2 = 15241383936$. The number is too large to be stored in a 32-bit integer, so the multiplication in the first statement fails compilation. The fact that a is 64-bit does not help unless we also specify the data type of the literal.
3.3 Scope of variables

The scope of a variable is where its value is kept, and can be referred in the code. Scopes in VB.NET are classified into four types: **block scope**, **procedure scope**, **module scope** and namespace scope. (Namespace scope is out of the scope of this book.)

The rules of scoping are different in different programming languages. Some programming languages do not have block scope at all, e.g. Pascal and **var** (but not **let**) keyword in Javascript.

Local variables

A **local variable** is a variable declared inside a **Sub** or **Function**. A local variables’ scope is its enclosing block. It can be used from the **Dim** statement to the last statement of the enclosing block.

If the enclosing block is a **Sub** or **Function**, it has procedure scope. Otherwise, it has block scope.

Make the scope of variables as narrow as possible. If the scope is too wide, there is more chance for your code access a wrong variable. These errors may not be detected during compilation, and are generally very difficult to fix.

The following example is a program that cannot be compiled because variable **a** is used before it is defined:

```vbnet
Console.WriteLine(a)  ' Compile error: "a" is declared after this statement.
Dim a As Integer = 10
```
And here is a program that cannot be compiled because the variable `value` is used outside of its scope.

```vbnet
Do
    Dim value As Integer = Console.ReadLine()
Loop While value <> -1 ' Compile error: "value" is not accessible here.
```

The program above can be corrected into one of the following:

```vbnet
Dim value As Integer
Do
    value = Console.ReadLine()
Loop While value <> -1
```

```vbnet
Do
    Dim value As Integer = Console.ReadLine()
    If value = -1 Then
        Exit Do
    End If
    ' Do
    Loop
```

**Variable counter in For...Next statement**

A block scoped counter variable is created with a `For...Next` statement in the following cases:

- if the variable is not already defined, or
- when the data type of the variable is defined with keyword `As`. In this case, the variable is declared in the same way as `Dim` statement.

💡 Do always define the data type of the variable in a `For...Next` statement.
Module scope

A variable has module scope if its enclosing block is a Module or Class. (Class is out of the scope of this book.)

Module scope variables can be referenced throughout the module, not just by the code after the declaration. Therefore, the following code is valid:

```vbnet
Module Module1
    Sub Main()
        Console.WriteLine(a) ' Prints "3".
    End Sub

    Dim a As Integer = 3
End Module
```

Variables in a Module block has its value initialised at the beginning of execution of the program. The value of this variable is also kept for the duration of the whole program. This is similar to global variables in many programming languages.
Shadowing of variables

In VB.NET, local variables **shadow** a variable with module scope. This means that the variable with module scope is hidden by the local variable. The variable with module scope is hidden for the whole block, not just for the part after the local variable is declared.

The hidden module scoped variable can be referred by qualifying its name (e.g. writing `Module1.a` instead of `a`). See the example below for details.

However, it is impossible to shadow a local variable. Code that attempts to do this (with `Dim` statement or `For ... Next` statement) fails compilation with the message “**Variable ‘[nameOfVariable]’ hides a variable in an enclosing block.**”

Here are a few examples:

```vbnet
Module Module1
    Dim a As Integer = 3

    Sub Main()
        Console.WriteLine(a) ' Prints "3".
        If a = 3 ' Evaluates to True.
            Console.WriteLine(a) ' Compile error.
        Dim a As Integer = 5
        Console.WriteLine(a) ' Prints "5".

        ' "Module1.a" refers to the "a" inside "Module1".
        Console.WriteLine(Module1.a) ' Prints "3".

        If a = 5 ' Evaluates to True.
            Dim a As Integer = 7 ' Compile error.
        End If
    End Sub
End Module
```
If you code properly, you will have compile errors thrown at you every now and then. This is actually good because compile errors help you to spot bugs early.

```vbnet
Module Module1
    Dim a As Integer = 3

    Sub Main()
        For a As Integer = 3 To 8 ' New local variable "a".
            Console.WriteLine(a)
        Next a
        Console.WriteLine(a) ' Prints "3".

        For a = 3 To 8 ' Use "a" in "Module1". Compile warning.
            Console.WriteLine(a)
        Next a
        Console.WriteLine(a) ' Prints "9".

        For b As Integer = 3 To 8 ' Compile error.
            Console.WriteLine(b)
        Next b
        Dim b As Integer = 4 ' This statement causes the above compile error.
    End Sub
End Module
```
Parameters in Sub and Function also shadow variables in module scope. Here is an example:

```vbnet
Module Module1
    Dim x As Integer = 3

    Sub a(x As Integer)
        Console.WriteLine(x)
        ' Isn't the following statement useless?
        x = x - 1 ' Refers to parameter "x" in Sub "a".
    End Sub

    Sub Main()
        Console.WriteLine(x) ' Prints "3".
        x = x + 1 ' Increase x to 4.

        a(1) ' Prints "1".
        a(2) ' Prints "2".

        Console.WriteLine(x) ' Prints "4".
    End Sub
End Module
```

**Lifetime of variables**

The lifetime of variables is more or less the same as the scope of the variable: if the control exits the scope of the variable, the content of the variable is lost.

In VB.NET, the lifetime of block scoped variables are however defined to be the enclosing procedure instead of the enclosing block. This has some implications if the variable is declared without initialisation. However, we should not be reading the value of an uninitialised variable anyway.

(Note: In VB.NET, if a variable is declared without initialisation, it is set to \(0\), \"\" or False at the beginning of its lifetime. In other programming languages, reading such values can lead to undefined behavior or compile failures.)
3.4 Best practices with variables

Here are some general guidelines using variables. Following the guidelines in your code make the code more readable and less prone to errors.

Naming of variables

- Use expressive names with complete words. No abbreviations unless the abbreviations is widely accepted.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>s</td>
<td>What is s? This can be “sum”, “string” or other things.</td>
</tr>
<tr>
<td>num</td>
<td>no</td>
<td>num is a widespread convention. However, no can be the opposite of “yes”.</td>
</tr>
<tr>
<td>numOfItems</td>
<td>items</td>
<td>items refer to an array of items, not a number.</td>
</tr>
<tr>
<td>i</td>
<td>-</td>
<td>OK if used as the counter in a loop.</td>
</tr>
<tr>
<td>x, y</td>
<td>-</td>
<td>OK if used as the coordinates of a point.</td>
</tr>
</tbody>
</table>

- Use camel case, i.e. capitalise the first letter of each word.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>numOfItems</td>
<td>numofitems</td>
</tr>
<tr>
<td>isUpperCase</td>
<td>isuppercase</td>
</tr>
</tbody>
</table>

- Boolean variables should be either a past participle, or a noun prepended with is or has. e.g. done, finished, isPrime, isDigit, hasData.

Most of these guidelines applies to naming procedures as well. However, it is recommended for procedures to start with a capital letter, e.g. IsPrime() and PrintResult().
Scope of variables

- Declare variables with the narrowest possible scope.
- If possible, initialise the variable at the same statement as the declaration.
- Do not try to read the value of an uninitialised variable. VB.NET allows you to do this, but some unexpected results may happen if the variable has block scope.
- Define and initialise a variable just before its value is used.

Data type of variables

- Unless you have a strong reason, declare a variable with its type specified.
- Do not store numerical values in String. This is inefficient.
Glossary

**counter**
A variable that keeps record of the iterations of a loop.

**debugger**
A computer program that is used to test and debug programs.

**global variable**
A variable that can be used in the whole program.

**iteration**
When a block of statements is executed multiple times, each repetition of these statements is called an iteration.

**lifetime**
The scope of a variable is where value of the variable is kept.

**literal**
A notation to represent a fixed value in source code.

**local variable**
A variable that is confined to a procedure, or a block inside the procedure.

**scope**
The scope of a variable is where the variable is visible.

**variable shadowing**
The mechanism that one variable hides another variable with the same name.