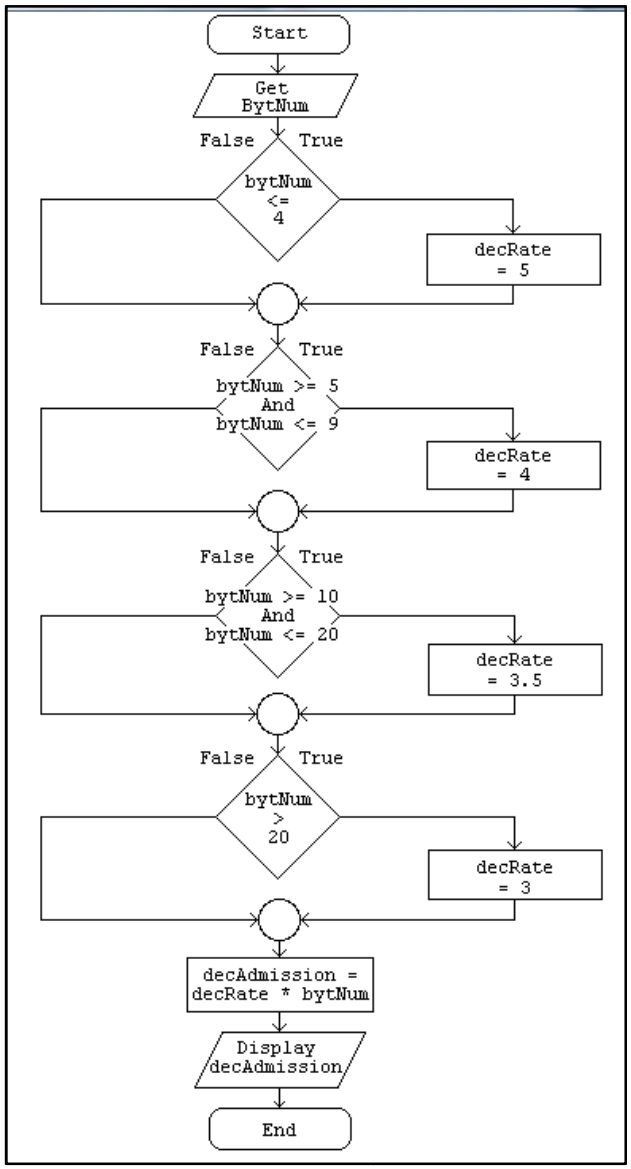
Selection Structures

There are two selection structures, code that selects the next block of code to execute: IF structures and Case structures.

IF STRUCTURE: There are many ways to test variables to decide which code to perform next. In the book, you've read about stacked IF, nested IF, and IF-ElseIF. Look at each one more closely.

Stacked IF statements

Each IF test must be performed, even after finding the true test.




```
'Declarations
Dim bytNum As Byte
Dim decRate As Decimal
Dim decAdmission As Decimal
'Input
bytNum = nudNum.Value

'Processing
If bytNum <= 4 Then
    decRate = 5
End If

If bytNum >= 5 And bytNum <= 9 Then
    decRate = 4
End If

If bytNum >= 10 And bytNum <= 20 Then
    decRate = 3.5
End If

If bytNum > 20 Then
    decRate = 3
End If
decAdmission = decRate * bytNum
'Output
lblAdmission.Text = decAdmission.ToString("c")
```

Desk Check with bytNum = 3
If bytNum <= 4 == TRUE
decRate = 5
If bytNum >= 5 and bytNum <= 9 == FALSE
If bytNum >= 10 and bytNum <= 20 == FALSE
If bytNum > 20 == FALSE

If will do 4 tests no matter which one is true.

Computers are very fast and can perform as many tests as it takes to achieve the result. But, efficient use of the computer's processing speed and memory is best programming practice, and can be critically important for small, embedded or specialty computers where speed and memory are not as readily available as in most desktop or laptop computers. In this code, there are 2 tests on two lines, such as:

```
If bytNum >= 5 AND if bytNum <= 9
```

The computer must do two separate operations, one for each operation, to determine if the result is true or false, and then it must do a third operation to determine if the AND operation is true or false. In the desk check for bytNum = 3, that line of code requires this processing:

```
bytNum >= 5 == FALSE
bytNum <= 9 == FALSE
FALSE and FALSE == FALSE
```

Three operations in that one line of code, three lines of code that don't need to execute because the true test has been found in the first line – you can see this is wasteful of the computer's resources.

Consider this version of the code, with only one test in each statement:

```
If bytNum <= 4 then
        decRate = 5
end if
if bytNum <= 9 then
        decRate = 4
end if
if bytNum <= 20 then
        decRate = 3.5
end if
if bytNum > 20 then
        decRate = 3
end if
```

Desk Check with bytNum = 3

```
If bytNum <= 4 == TRUE, so
        decRate = 5
If bytNum <= 9 == TRUE, so
        decRate = 4
If bytNum <= 20 == TRUE, so
        decRate = 3.5
If bytNum > 20 == FALSE
```

It performs 4 tests, but changes the decRate 3 times, because the result is true in 3 of the tests. The final version of decRate is incorrect, so this is a logic error.

It is possible to change this code so that you will only have one true test.
Consider this version, with only test in each statement but with the tests in a different order:

```
If bytNum > 20 then
        decRate = 3
end if
if bytNum >= 10 then
        decRate = 3.5
end if
if bytNum >= 5 then
        decRate = 3
end if
if bytNum <= 4 then
        decRate = 5
end if
```

Desk Check with bytNum = 3

```
If bytNum >= 20 == FALSE
If bytNum >= 10 == FALSE
If bytNum >= 5 == FALSE
If bytNum <= 4 == TRUE
        decRate = 5
```

It performs 4 tests, but does not change the decRate 3 times as in the previous version. The order of the tests makes it possible to use only one test in each statement and get the correct answer.

Stacked IF structures make sense when different variables are being tested. For example, in looking for the largest of 3 values, the sample program showed one IF structure that tests if value 1 is greater than value 2, and assigns the larger one to a different variable for the largest value. The stacked IF that follows tests if value 3 is larger than the largest value. Those are different variables, a different IF structure is appropriate here.

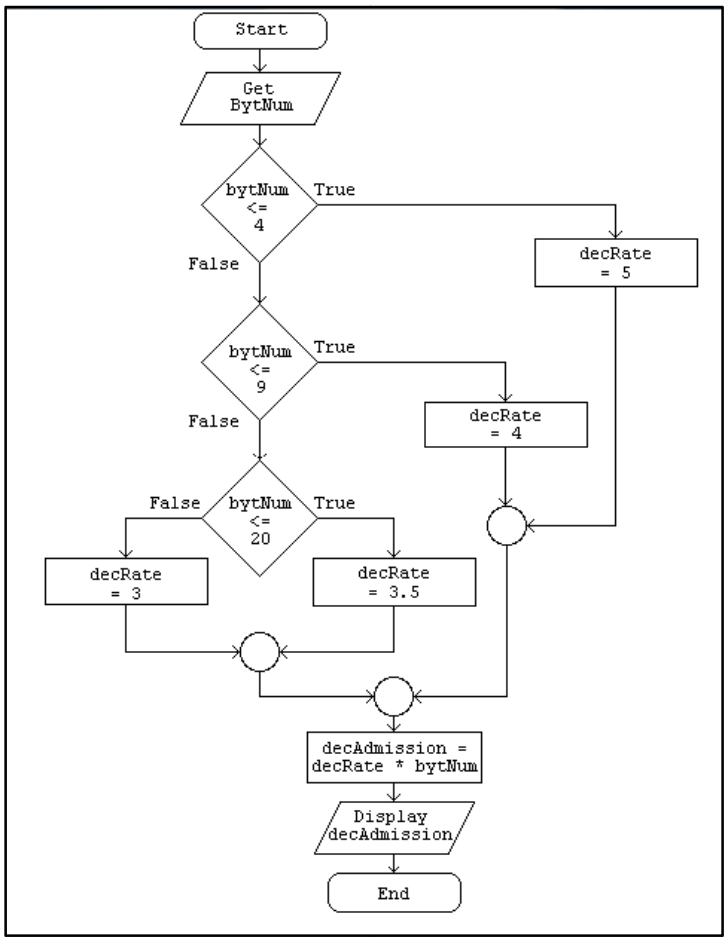
IF-ElseIF STRUCTURES

The format of this structure looks like this:

IF (test) then
    ... code ...
Else IF (test) then
    ... code ...
(as many Else-IFs as needed can go here)
(an ELSE can also go here, executes when all the other tests were false)
End If

There is only one End If because this is a single structure. Stacked and nested IF structures are multiple blocks of code; keeping track of where one ends can be tricky.

In an IF-ElseIf structure, once the true test is found, the computer skips everything else in the structure. It won't perform any more of the tests listed, which can be far more efficient than seen above.




```
'Declarations
Dim bytNum As Byte
Dim decRate As Decimal
Dim decAdmission As Decimal

'Input
bytNum = nudNum.Value

'Processing
If bytNum <= 4 Then
     decRate = 5
ElseIf bytNum <= 9 Then
     decRate = 4
ElseIf bytNum <= 20 Then
     decRate = 3.5
Else
     decRate = 3
End If

decAdmission = decRate * bytNum

'Output
lblAdmission.Text = decAdmission.ToString("c")
```

Desk Check with bytNum = 3
If bytNum <= 4 == TRUE
    decRate = 5
jump to End If and leave the structure
This resulted in many less operations!

In this version, it doesn't matter that the tests bytNum <= 9 or bytNum <= 20 would also be true, because those tests are never executed. An IF-ElseIF structure may execute as many tests as a stacked IF structure, if the final test is the true one. But it may also execute only one test, if that turns out to be the true test.

As with stacked IF structures, you can change the order of the tests to eliminate multiple tests on one line. Consider a program that determines the letter grade associated with a test score. For scores 90 and above, the letter grade is A; for scores between 80 and 89, the letter grade is B; and so on. You can specify the full range, the maximum and minimum values, in the IF test like this:

```
If testScore >= 90 and testScore <= 100 then
        letterGrade = "A"
Else If testScore >= 80 and testScore < 90 then
        letterGrade = "B"
```

and so on. Or, you can test a little smarter than that with a single test, if you understand the implications. Consider this as the first test:

```
If testScore >= 90
```

If this is not true, that means the score is not 90, 91, 92, and so on. It is not in the 90s at all. So if the next test is this:

```
Else If testScore >= 80
```

There is an implication that if the computer executes this test, it's looking for a score between 80 and 90. We already know the score is not greater than 90, because the first test failed.

Consider this code:

```
If testScore >= 60 then
        letterGrade = "D"
else if testScore >= 70 then
        letterGrade = "C"
else if testScore >= 80 then
        letterGrade = "B"
else if testScore >= 90 then
        letterGrade = "A"
else
        letterGrade = "F"
end if
```

```
Desk Check with the testScore = 72
If testScore >= 60 == TRUE
        letterGrade = "D"
Find the End If and exit
Desk Check with the testScore = 92
If testScore >= 60 == TRUE
        letterGrade = "D"
Find the End If and exit
Both of these are incorrect, there is a logic error
in this order of tests.
```

The IF-ElseIF structure can be much more efficient than stacked or nested IF structures. Be sure you desk check the results to verify the order of the tests and the implied boundaries are correct.

DOING THE MATH

When the computer tests if a condition is true, it has to **resolve** the value of the expressions. The expressions use relational operators – what is the relationship between these two things? There are 6 relational operators – equal (=), not equal (<>), greater than (>), less than (<), greater than or equal to

(=>), less than or equal to (<=). The relationship between two things can always be described by 3 of these operators. Consider the relationships between the numbers 10 and 15:

(1) 10 is less than 15;

(2) 10 is less than or equal to 15;

(3) 10 is not equal to 15.

The result of these relationships always boils down to true or false.

It's not difficult to determine how a single relational operation resolves. With more complicated logic, you need to work through each piece to come up with the final resolution. Of course, the mathematical rules for order of operations apply. For example, assume x = 20 and y = 15, and resolve this equation:

```
If ( (x * 2) + (y * 2) < 100 )      :: replace x and y with their values
If ( (20 * 2) + (15 * 2) < 100)     :: do the math inside parentheses first
If ( (40) + (30) < 100)             :: do the addition
If ( 70 < 100)                      :: this resolves to true
```

Often the relationships to be resolved are more complicated. We might say "if it's raining and if it's windy, I'll put on a raincoat". The word "AND" connects 2 conditions – the first is whether it's raining, the second is whether it's windy. This is a logical operation, and the most frequently used **logical operators** are AND and OR. Again, these must resolve to true or false. An AND operation will resolve to true if both sides of the statement are true; an OR operation resolves to true if at least one side of the statement is true. These are laid out in truth tables:

AND	Condition 1 = True	Condition 1 = False
Condition 2 = True	True	False
Condition 2 = False	False	False

If condition 1 is true AND condition 2 is true, the test resolves to true. If either condition is false, the overall test is false.

OR	Condition 1 = True	Condition 1 = False
Condition 2 = True	True	True
Condition 2 = False	True	False

In OR tests, if condition 1 is true, the overall test resolves to true. If condition 2 is true, the overall test resolves to true. If both are true, the overall test resolves to true.

If you were trying to validate that a user entered a number between 1 and 10 in variable intValue, you can use the AND logical operation like this:

```
If ( intValue > 0 ) AND ( intValue <= 10 )
```

For this IF test to resolve to true, it must be true that the number is greater than 0 AND the number must be less than or equal to 10. Both conditions must be true since they are connected with the AND operator for the whole sentence to be true. If either one is false, the overall IF test resolves to false.

Note that there are multiple ways to write the tests to check the same information. The following are equivalent tests:

If (intValue > 0) AND (intValue < 11)
If (intValue > 0) AND (intValue <= 10)
If (intValue => 1) AND (intValue < 11)
If (intValue => 1) AND (intValue <= 10)

If you wanted to check that a user entered a 1 or a 2, but not any other number, you can use the OR logical operation like this:

If (intValue = 1) OR (intvalue = 2)

The overall IF test resolves to true if either one of those conditions is true. Both can be true as well. As you can see from the truth table above, both conditions would have to be false for the OR test to be false.

You can have multiple logical operators in one test. Here are some examples:

If (intValue = 1) OR (intValue = 2) OR (intValue = "Q")
If (intValue > 0 AND intValue < 11) OR (total > 100)

CASE STRUCTURES: The IF structure can test multiple variables using the logical operators AND and OR. Case structures test a single variable, but test more values of that single variable. It is possible to create the same results with both an IF and a Case structure, when there is only one variable in the tests. It may not be possible to create a Case structure that does the same work as an IF structure, if there are multiple variables or logical operations, such as those examples just above.

The Case structure begins with the variable to test:

Select Case intValue (you can read this as "Select the current case of this variable")

The tests of the Case structure can be written in several ways. The following are all equivalent, for an integer variable "intValue" that may have values 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Case 1 TO 10
Case 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

You may use relational operators as well.

Case < 11
Case <= 10
Case > 0 – but this will allow any number greater than 0, so it could include 23 or 17

Note that you cannot say put this kind of case test in one line: Case > 0 AND Case < 11
Instead, use one of the choices above.

Summary: IF tests can check multiple variables; Case checks a number of values of a single variable.