To be able to construct a simple logical (Boolean) expression to evaluate a given condition.

To be able to construct a complex Boolean expression to evaluate a given condition.

To be able to construct an If-Then-Else statement to perform a specific task.

To be able to construct an If statement to perform a specific task.

To be able to construct a series of nested If statements to perform a specific task.

To be able to describe an algorithm walk-through and tell how it is used.

To be able to explain the purpose of tracing the execution of Visual Basic code.

To be able to test and debug a Visual Basic program.
So far, the statements in our programs have been executed in the same order in which we write them, except when an event occurs. The first statement is executed, then the second, and so on. The method call and the event handler, which execute a separate sequence of statements, provide variations of this ordering. But what if we want the computer to execute the statements in an order other than sequentially? Suppose we want to check the validity of input data and then perform a calculation or display an error message, but not both. To do so, we must be able to ask a question and then, based on the answer, choose one or another course of action.

The If statement allows us to execute statements in an order that is different from their physical order. We can ask a question with it and do one thing if the answer is yes (true) or another if the answer is no (false). In the first part of this chapter, we deal with asking questions; in the second part, we deal with the If statement itself.

6.1 Flow of Control

The order in which statements are executed in a program is called the flow of control. In a sense, the computer is under the control of one statement at a time. When a statement has been executed, control is turned over to the next statement (like a baton being passed in a relay race).

Flow of control is normally sequential (see Figure 6.1). That is, when one statement is finished executing, control passes to the next statement in the program. If you want the flow of control to be nonsequential, you can use control structures, special statements that transfer control to a statement other than the one that physically comes next. You have already seen
that method calls, especially event handlers, are control structures that alter the flow of control so that a separate sequence of statements can be executed.

**Selection**

You can use a selection (or branching) control structure when you want the computer to choose between alternative actions. You make an assertion, a claim that is either true or false. If the assertion is true, the computer executes one statement. If it is false, it executes another (see Figure 6.2). The computer’s ability to solve practical problems is a product of its ability to make decisions and execute different sequences of instructions.

The Payroll program in Chapter 1 shows the selection process at work. The computer must decide whether or not a worker has earned overtime pay. It does this by testing the assertion that the person has worked over 40 hours. If the assertion is true, the computer follows the instructions for computing overtime pay. If the assertion is false, the computer simply computes the regular pay. Before examining selection control structures in Visual Basic, let’s look closely at how we get the computer to make decisions.
6.2 Conditions and Logical Expressions

To ask a question in Visual Basic, we don’t phrase it as a question; we state it as an assertion. If the assertion we make is true, the answer to the question is yes. If the assertion is not true, the answer to the question is no. For example, if we want to ask, “Are we having spinach for dinner tonight?” we would say, “We are having spinach for dinner tonight.” If the assertion is true, the answer to the question is yes. If not, the answer is no.

So, asking questions in Visual Basic means making an assertion that is either true or false. The computer evaluates the assertion, checking it against some internal condition (the values stored in certain variables, for instance) to see whether it is true or false.

The Boolean Data Type

The Boolean data type consists of just two values, the constants True and False. The reserved word Boolean is pronounced bool-e-un. Boolean data is used for testing conditions in a program so that the computer can make decisions (as in a selection control structure).

We declare variables of type Boolean the same as we declare variables of other standard types, by writing the Dim keyword, followed by the variable name, the As keyword, and the word Boolean:

```vbnet
Dim dataOK As Boolean
Dim done As Boolean
Dim taxable As Boolean
```

Each variable of type Boolean can contain one of two values: True or False. It’s important to understand right from the beginning that True and False are not variable names and they are not strings. They are special constants in Visual Basic and, in fact, are reserved words.

Logical Expressions

In programming languages, assertions take the form of logical expressions (also called Boolean expressions). Just as an arithmetic expression is made up of numeric values and operations, a logical expression is made up of logical values and operations. Every logical expression has one of the two Boolean values: True or False.

Here are some examples of logical expressions:

- A Boolean variable or constant
- An arithmetic expression followed by a relational operator followed by an arithmetic expression
- A logical expression followed by a logical operator followed by a logical expression

---

1The name boolean is a tribute to George Boole, a nineteenth-century English mathematician who described a system of logic using variables with just two values, True and False. (See the Background Information box on page 222.)
Let’s look at each of these in detail.

**Boolean Variables and Constants** As we have seen, a Boolean variable is a variable declared to be of type Boolean, and it can contain either the value True or the value False. For example, if dataOK is a Boolean variable, then

```
dataOK = True
```

is a valid assignment statement.

**Relational Operators** Another way of assigning a value to a Boolean variable is to set it equal to the result of comparing two expressions with a relational operator. Relational operators test a relationship between two values.

Let’s look at an example. In the following program fragment, lessThan is a Boolean variable and i and j are Integer variables:

```
lessThan = (i < j) ' Compare i and j with the "less than" relational operator, and assign the value to lessThan
```

By comparing two values, we assert that a relationship (such as “less than”) exists between them. If the relationship does exist, the assertion is true; if not, it is false. These are the relationships we can test for in Visual Basic:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relationship Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>

An expression followed by a relational operator followed by an expression is called a relational expression. The result of a relational expression is of type Boolean. For example, if x is 5 and y is 10, the following expressions all have the value True:

```
x <> y
y > x
x < y
y >= x
x <= y
```

If x is the character "M" and y is the character "R", the values of the expressions are still True because the relational operator <, used with letters, means “comes before in the alphabet,” or, more properly, “comes before in the collating sequence of the character
set.” For example, in the ASCII subset of the Unicode character set, all of the uppercase letters are in alphabetical order, as are the lowercase letters, but all of the uppercase letters come before the lowercase letters. So

"M"c < "R"c

and

"m"c < "r"c

have the value True, but

"m"c < "R"c

has the value False.

Of course, we have to be careful about the data types of things we compare. The safest approach is always to compare identical types: Integer with Integer, Double with Double, Char with Char, and so on. If you mix data types in a comparison, implicit type conversion can take place just as in arithmetic expressions. If an Integer value and a Double value are compared, the computer temporarily converts the Integer value to its Double equivalent before making the comparison. As with arithmetic expressions, it’s wise to use type conversion functions to make your intentions known:

someDouble >= CDbl(someInt)

If you try to compare a Boolean value with a numeric value (probably by mistake), the compiler gives you an error message. Values of type Boolean cannot be converted to any type other than String. When a Boolean variable is concatenated with a string, its value is automatically converted to either "True" or "False". No type can be converted to Boolean.

Be careful to compare Char values only with other Char values. For example, the comparisons

"0"c < "9"c

and

0 < 9

are appropriate, but comparing a digit in quotes (a character) and a digit such as

"0"c < 9

generates an error by the compiler.
You can use relational operators not only to compare variables or constants, but also to compare the values of arithmetic expressions. The following table shows the results of using the relational operators to compare expressions made up of adding 3 to \( x \) and multiplying \( y \) by 10 for different values of \( x \) and \( y \):

<table>
<thead>
<tr>
<th>Value of ( x )</th>
<th>Value of ( y )</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>2</td>
<td>( x + 3 \leq y * 10 )</td>
<td>True</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>( x + 3 \leq y * 10 )</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>( x + 3 \neq y * 10 )</td>
<td>False</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>( x + 3 = y * 10 )</td>
<td>True</td>
</tr>
<tr>
<td>100</td>
<td>5</td>
<td>( x + 3 &gt; y * 10 )</td>
<td>True</td>
</tr>
</tbody>
</table>

**Comparing Strings** Unlike many other computer languages, you can compare strings with the relational operators in Visual Basic and expect to receive reliable results. We can also compare strings with a set of value-returning instance methods that Visual Basic supplies as part of the `String` class. Because they are instance methods, the method name is written following a `String` object, separated by a dot. The string that the method name is appended to is one of the strings in the comparison, and the string in the parameter list is the other. Because there are times we want to compare strings ignoring capitalization, the `String` class provides methods called `ToLower` and `ToUpper` that convert all the characters of a string to lowercase or uppercase, respectively. The two most useful comparison methods are summarized in the following table, along with `ToLower` and `ToUpper`.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Parameter Type</th>
<th>Returns</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals</td>
<td>String</td>
<td>Boolean</td>
<td>Tests for equality of string contents.</td>
</tr>
<tr>
<td>CompareTo</td>
<td>String</td>
<td>Integer</td>
<td>Returns 0 if equal, a positive integer if the string in the parameter comes before the string associated with the method, and a negative integer if the parameter comes after it.</td>
</tr>
<tr>
<td>ToLower</td>
<td>String</td>
<td></td>
<td>Returns a new identical string, except the characters are all lowercase.</td>
</tr>
<tr>
<td>ToUpper</td>
<td>String</td>
<td></td>
<td>Returns a new identical string, except the characters are all lowercase.</td>
</tr>
</tbody>
</table>

For example, if `lastName` is a `String` variable, you can write

```
lastName.Equals("Olson")   ' Tests whether lastName equals "Olson"
```
Because every String literal is also a String object, Visual Basic lets you append the method call to a literal, if you so choose.

"Olson".Equals(lastName)  ' Tests whether lastName equals "Olson"

As another example, you might write

0 > lastName.CompareTo("Olson")  ' Tests if lastName comes before "Olson"

Comparison of strings follows the collating sequence of the Unicode character set. When the computer tests a relationship between two strings, it begins with the first character of each, compares them according to the collating sequence, and if they are the same, repeats the comparison with the next character in each string. The character-by-character test proceeds until a mismatch is found, the final characters have been compared and are equal, or one string runs out of characters. If all their characters are equal, then the two strings are equal. If a mismatch is found, then the string with the character that comes before the other is the "lesser" string. If their characters are equal throughout the shorter string, the shorter string comes before the longer string.

For example, given the statements

Dim word1 As String
Dim word2 As String

word1 = "Tremendous"
word2 = "Small"

the following relational expressions have the values shown.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>word1.Equals(word2)</td>
<td>False</td>
<td>They are unequal in the first character.</td>
</tr>
<tr>
<td>0 &lt; word1.CompareTo(word2)</td>
<td>True</td>
<td>&quot;T&quot; comes after &quot;S&quot; in the collating sequence.</td>
</tr>
<tr>
<td>0 &lt; word1.CompareTo(&quot;small&quot;)</td>
<td>False</td>
<td>&quot;T&quot; comes before &quot;s&quot; in the collating sequence.</td>
</tr>
<tr>
<td>0 &gt; word1.CompareTo(&quot;Tremble&quot;)</td>
<td>False</td>
<td>Fifth characters don't match, and &quot;b&quot; comes before &quot;e&quot;.</td>
</tr>
<tr>
<td>word2.Equals(&quot;Small&quot;)</td>
<td>True</td>
<td>They are equal.</td>
</tr>
<tr>
<td>0 = &quot;cat&quot;.CompareTo(&quot;dog&quot;)</td>
<td>False</td>
<td>They are unequal.</td>
</tr>
</tbody>
</table>

In most cases, the ordering of strings corresponds to alphabetical ordering. But when strings have mixed case letters, you can get nonalphabetical results. For example, in a phone book we expect to see Macauley before MacPherson, but the Unicode collat-
ing sequence places all English uppercase letters before the lowercase letters, so the string "MacPherson" compares as less than "Macauley". To compare strings for strict alphabetical ordering, all the characters must be in the same case. Here we see examples in which ToLower and ToUpper are used to perform the comparison correctly.

```
lowerCaseString = myString.ToLower()
upperCaseString = myString.ToUpper()
```

You can use these methods directly in a comparison expression. For example, the following expressions convert `word1` and `word2` to the same case before comparing them. It doesn’t matter whether the strings are both converted to uppercase or both are converted to lowercase, as long as they are the same.

```
0 > word1.ToLower().CompareTo(word2.ToLower())
0 > word1.ToUpper().CompareTo(word2.ToUpper())
```

If two strings with different lengths are compared and the comparison is equal up to the end of the shorter string, then the shorter string compares as less than the longer string. For example, if `word2` contains "Small", the expression

```
0 > word2.CompareTo("Smaller")
```

yields `True` because the strings are equal up to their fifth character position (the end of the string on the left) and the string on the right is longer.

Logical Operators In mathematics, the logical (or Boolean) operators AND, OR, and NOT take logical expressions as operands. Visual Basic uses these same words (`And`, `Or`, and `Not`) for the logical operators. By combining relational operators with logical operators, you can make more complex assertions. For example, suppose you want to determine whether a final score is greater than 90 and a midterm score is greater than 70. In Visual Basic you would write the expression this way:

```
finalScore > 90 And midtermScore > 70
```

The `And` operation requires both relationships to be true in order for the overall result to be true. If either or both of the relationships is `False`, the entire result is `False`.

The `Or` operation takes two logical expressions and combines them. If `either or both` are true, the result is `True`. Both values must be `False` for the result to be `False`. Now you can determine whether the midterm grade is an A or the final grade is an A. If either the midterm grade or the final grade equals A, the assertion is `True`. In Visual Basic, you can write the expression like this:

```
midtermGrade = "A"c Or finalGrade = "A"c
```
The And and Or operators always appear between two expressions; they are binary (two-operand) operators. The Not operator is a unary (one-operand) operator. It precedes a single logical expression and gives its opposite as the result. If \((\text{grade} = \text{"A"})\) is false, then \(\text{Not(grade} = \text{"A"})\) is true. Not gives a convenient way of reversing the meaning of an assertion. For example,

\[\text{Not(hours > 40)}\]

is the equivalent of

\[\text{hours <= 40}\]

In some contexts, the first form is clearer; in others, the second makes more sense.

The following pairs of expressions are equivalent:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{Not(a = b)})</td>
<td>(a &lt;&gt; b)</td>
</tr>
<tr>
<td>(\text{Not(a = b Or a = c)})</td>
<td>(a &lt;&gt; b \text{ And } a &lt;&gt; c)</td>
</tr>
<tr>
<td>(\text{Not(a = b And c &gt; d)})</td>
<td>(a &lt;&gt; b \text{ Or } c &lt;= d)</td>
</tr>
</tbody>
</table>

Take a close look at these expressions to be sure you understand why they are equivalent. Try evaluating them with some values for \(a\), \(b\), \(c\), and \(d\). Notice the pattern: The expression on the left is just the one to its right with Not added and the relational and logical operators reversed (for example, \(=\) instead of \(<>\) and Or instead of And). Remember this pattern. It allows you to rewrite expressions in the simplest form.\(^2\)

Logical operators can be applied to the results of comparisons. They also can be applied directly to variables of type Boolean. For example, instead of writing

\[\text{IsElector} = (\text{age} >= 18 \text{ And district} = 23)\]

to assign a value to the Boolean variable \(\text{isElector}\), you could use two intermediate Boolean variables, \(\text{isVoter}\) and \(\text{isConstituent}\):

\[\text{isVoter} = (\text{age} >= 18)\]
\[\text{isConstituent} = (\text{district} = 23)\]
\[\text{isElector} = \text{isVoter And isConstituent}\]

The two tables that follow summarize the results of applying And and Or to a pair of logical expressions (represented here by Boolean variables \(x\) and \(y\)).

\(^2\)In Boolean algebra, the pattern is formalized by a theorem called DeMorgan's law.
### 6.2 Conditions and Logical Expressions

<table>
<thead>
<tr>
<th>Value of $x$</th>
<th>Value of $y$</th>
<th>Value of $(x \text{ And } y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value of $x$</th>
<th>Value of $y$</th>
<th>Value of $(x \text{ Or } y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

The following table summarizes the results of applying the Not operator to a logical expression (represented by Boolean variable $x$).

<table>
<thead>
<tr>
<th>Value of $x$</th>
<th>Value of Not($x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

**Short-circuit Evaluation**

Consider the logical expression

$$i = 1 \text{ And } j > 2$$

Some programming languages use *full evaluation* of logical expressions. With full evaluation, the computer first evaluates both subexpressions (both $i = 1$ and $j > 2$) before applying the And operator to produce the final result.

In contrast, Visual Basic uses *short-circuit (or conditional) evaluation* of logical expressions. Evaluation proceeds from left to right, and the computer stops evaluating subexpressions as soon as possible—that is, as soon as it knows the Boolean value of the entire expression. How can the computer know if a lengthy logical expression yields True or False if it doesn’t examine all the subexpressions? Let’s look first at the And operation.

An And operation yields the value True only if both of its operands are True. In the previous expression, suppose that the value of $i$ happens to be 95. The first subexpression yields False, so it isn’t necessary even to look at the second subexpression. The computer stops evaluation and produces the final result of False.
With the Or operation, the left-to-right evaluation stops as soon as a subexpression yielding True is found. Remember that an Or produces a result of True if either one or both of its operands are True. Given this expression:

\[ c \leq d \text{ Or } e = f \]

if the first subexpression is True, evaluation stops and the entire result is True. The computer doesn’t waste time evaluating the second subexpression.

---

**Background Information**

**George Boole**

Boolean algebra is named for its inventor, English mathematician George Boole, born in 1815. His father, a tradesman, began teaching him mathematics at an early age. But Boole initially was more interested in classical literature, languages, and religion—interests he maintained throughout his life. By the time he was 20, he had taught himself French, German, and Italian. He was well-versed in the writings of Aristotle, Spinoza, Cicero, and Dante, and wrote several philosophical papers himself.

At 16, to help support his family, he took a position as a teaching assistant in a private school. His work there and a second job left him little time to study. A few years later, he opened a school and began to learn higher mathematics on his own. In spite of his lack of formal training, his first scholarly paper was published in the *Cambridge Mathematical Journal* when he was just 24. Boole went on to publish over 50 papers and several major works before he died in 1864, at the peak of his career.

Boole’s *The Mathematical Analysis of Logic* was published in 1847. It would eventually form the basis for the development of digital computers. In the book, Boole set forth the formal axioms of logic (much like the axioms of geometry) on which the field of symbolic logic is built.

Boole drew on the symbols and operations of algebra in creating his system of logic. He associated the value 1 with the universal set (the set representing everything in the universe) and the value 0 with the empty set, and restricted his system to these two quantities. He then defined operations that are analogous to subtraction, addition, and multiplication. Variables in the system have symbolic values. For example, if a Boolean variable *P* represents the set of all plants, then the expression \( 1 - P \) refers to the set of all things that are not plants. We can simplify the expression by using \( -P \) to mean “not plants.” (\( -P \) is simply 0 because we can’t remove elements from the empty set.) The subtraction operator in Boole’s system corresponds to the Not operator in Visual Basic. In a Visual Basic program, we might set the value of the Boolean variable *plant* to True when the name of a plant is entered, and Not(*plant*) is True when the name of anything else is input.

(continued)
George Boole

The expression $0 + P$ is the same as $P$. However, $0 + P + F$, where $F$ is the set of all foods, is the set of all things that are either plants or foods. So the addition operator in Boole’s algebra is the same as the Visual Basic Or operator.

The analogy can be carried to multiplication: $0 \times P$ is 0, and $1 \times P$ is $P$, but what is $P \times F$? It is the set of things that are both plants and foods. In Boole’s system, the multiplication operator is the same as the And operator.

In 1854, Boole published *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*. In the book, he described theorems built on his axioms of logic and extended the algebra to show how probabilities could be computed in a logical system. Five years later, Boole published *Treatise on Differential Equations*, then *Treatise on the Calculus of Finite Differences*. The latter is one of the cornerstones of numerical analysis, which deals with the accuracy of computations. (In Chapter 13, we examine the important role numerical analysis plays in computer programming.)

Boole received little recognition and few honors for his work. Given the importance of Boolean algebra in modern technology, it is hard to believe that his system of logic was not taken seriously until the early twentieth century. George Boole is truly one of the founders of computer science.

Precedence of Operators

In Chapter 4 we discussed the rules of precedence, the rules that govern the evaluation of complex arithmetic expressions. Visual Basic’s rules of precedence also govern relational and logical operators. Here’s a list showing the order of precedence for the arithmetic, relational, and logical operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Highest precedence</td>
</tr>
<tr>
<td>Unary +</td>
<td></td>
</tr>
<tr>
<td>=   &lt;   &gt;   &lt;=  &gt;=</td>
<td></td>
</tr>
<tr>
<td>Not</td>
<td>Lowest precedence</td>
</tr>
<tr>
<td>And</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td></td>
</tr>
</tbody>
</table>

Operators on the same line in the list have the same precedence. If an expression contains several operators with the same precedence, most of the operators group (or associate) from left to right. For example, the expression

$$a / b * c$$
means \((a / b) \times c\), not \(a / (b \times c)\). However, the Not operator groups from right to left. Although you’d never have occasion to use this expression:

```
Not Not badData
```

the meaning is \(\text{Not} (\text{Not} (\text{badData}))\) rather than the meaningless \(\text{Not} (\text{Not} (\text{badData}))\).

Appendix B, Operator Precedence, lists the order of precedence for all operators in Visual Basic. In skimming the appendix, you can see that a few of the operators associate from right to left (for the same reason we just described for the Not operator).

Parentheses are used to override the order of evaluation in an expression. If you’re not sure whether parentheses are necessary, use them anyway. The compiler disregards unnecessary parentheses. So if they clarify an expression, use them. Some programmers like to include extra parentheses when assigning a relational expression to a Boolean variable:

```
dataInvalid = (inputVal = 0)
```

The parentheses are not needed in the example above. The compiler understands that since the first equal sign is meant to be an assignment operator, the second equal sign is being used as the equality operator. So we could write the statement as

```
dataInvalid = inputVal = 0
```

but the parenthesized version is more readable. Without the parentheses, it looks as if we are trying to assign the value 0 to both inputVal and dataInvalid.

One final comment about parentheses: Visual Basic, like other programming languages, requires that parentheses always be used in pairs. Whenever you write a complicated expression, take a minute to go through and pair up all of the opening parentheses with their closing counterparts.

---

PEANUTS reprinted by permission of United Features Syndicate, Inc.
Software Engineering Tip

Changing English Statements into Logical Expressions

In most cases, you can write a logical expression directly from an English statement or mathematical term in an algorithm. But you have to watch out for some tricky situations. Remember our sample logical expression:

\[
\text{midtermGrade} = "A"c \text{ Or finalGrade} = "A"c
\]

In English, you would be tempted to write this expression: "Midterm grade or final grade equals A." In Visual Basic, you can't write the expression as you would in English. That is,

\[
\text{midtermGrade Or finalGrade} = "A"c
\]

won't work because the Or operator is connecting a Char value (midtermGrade) and a logical expression (finalGrade = "A"c). The two operands of Or must be logical expressions. This example generates a syntax error message.

A variation of this mistake is to express the English assertion "i equals 3 or 4" as

\[
i = 3 \text{ Or } 4
\]

But the syntax is incorrect. In the second expression, 4 is an Integer rather than a Boolean value. The Or operator (and the And operator) can only connect two Boolean expressions. Here's what we want:

\[
i = 3 \text{ Or } i = 4
\]

In math books, you might see a notation like this:

\[
12 < y < 24
\]

which means "y is between 12 and 24." This expression is illegal in Visual Basic. First, the relation \(12 < y\) is evaluated, giving a Boolean result. The computer then tries to compare it with the number 24. Because a Boolean value cannot be converted to any type other than String, the expression is invalid. To write this expression correctly in Visual Basic, you must use the And operator as follows:

\[
12 < y \text{ And } y < 24
\]
Relational Operators with Floating-Point Types

So far, we’ve talked only about comparing Integer, Char, and String values. Here we look at Float and Double values.

Do not compare floating-point numbers for equality. Because small errors in the rightmost decimal places are likely to arise when calculations are performed on floating-point numbers, two Float or Double values rarely are exactly equal. For example, consider the following code that uses two Double variables named oneThird and x:

```
oneThird = 1.0 / 3.0
x = oneThird + oneThird + oneThird
```

We would expect x to contain the value 1.0, but it may not. The first assignment statement stores an approximation of 1/3 into oneThird, perhaps 0.333333. The second statement stores a value like 0.999999 into x. If we now ask the computer to compare x with 1.0, the comparison may yield False.

Instead of testing floating-point numbers for equality, we test for near equality. To do so, we compute the difference between the two numbers and test to see if the result is less than some maximum allowable difference. For example, we often use comparisons like this:

```
Math.Abs(r – s) < 0.00001
```

where Math.Abs is the absolute value method from the Visual Basic Math class. The expression Math.Abs(r – s) computes the absolute value of the difference between two variables r and s. If the difference is less than 0.00001, the two numbers are close enough to call them equal. We discuss this problem with floating-point accuracy in more detail in Chapter 13.

### 6.3 The If Statement

Now that we’ve seen how to write logical expressions, let’s use them to alter the normal flow of control in a program. The If statement is the fundamental control structure that allows branches in the flow of control. With it, we can ask a question and choose a course of action: if a certain condition exists (the assertion is true), perform one action, else perform a different action.

The computer performs just one of the two actions under any given set of circumstances. Yet we must write both actions into the program. Why? Because, depending on the circumstances, the computer can choose to execute either of them. The If statement gives us a way of including both actions in a program and gives the computer a way of deciding which action to take.

#### The If-Else Form

In Visual Basic, the If statement comes in two forms, the If-Else form and the If form. Let’s look first at the If-Else. Here is its syntax template:
The expression in parentheses must produce a *Boolean* result. At run time, the computer evaluates the expression. If the value is *True*, the computer executes Statement1A. If the value of the expression is *False*, Statement1B is executed. Statement1A is often called the *Then-clause*; Statement1B, the *Else-clause*. Figure 6.3 illustrates the flow-of-control of the *If-Else*. In the figure, Statement2 is the next statement in the program after the entire *If* statement.

The code fragment below shows how to write an *If* statement in a program. Observe the indentation of the *Then*-clause and the *Else*-clause, which makes the statement easier to read. Also notice the placement of the statement following the *If* statement.

```vbnet
If (hours <= 40R) Then
    pay = rate * hours
Else
    pay = rate * (40R + (hours - 40R) * 1.5)
End If
lblOut.Text = "Pay is " & pay
```

*Figure 6.3  If-Else flow of control*
In terms of instructions to the computer, the previous code fragment says, “If \( \text{hours} \) is less than or equal to 40.0, compute the regular pay and then go on to execute the output statements. But if \( \text{hours} \) is greater than 40, compute the regular pay and the overtime pay, and then go on to execute the output statement.” Figure 6.4 shows the flow of control of this If statement.

If-Else statements are often used to check the validity of input. For example, before we ask the computer to divide by a data value, we should be sure that the value is not zero. (Even computers can’t divide something by zero.) If the divisor is zero, our program should thus display an error message. Here’s an example that adds an error message to a label on a form.

```vba
Dim errorMsg As String
errorMsg = "Division by zero is not allowed. " & _
            "Enter a new value for divisor and press Done button."
If (divisor <> 0) Then
    result = dividend / divisor
Else
    lblOut.Text = errorMsg
End If
```

If this code fragment is within the handler for a button-controlling screen input, the program waits until another event occurs, either the pressing of the button or the closing of the input window.
Statement Sequences (Compound Statements)

In our division-by-zero example, suppose that when the divisor is equal to zero we want to do two things: add the error message to the form and set the variable named result equal to a special value like Integer.MaxValue. We would need two statements in the same branch, but the syntax template seems to limit us to one.

What we really want to do is turn the Else-clause into a sequence of statements. This is easy. Remember from Chapter 2 that the compiler treats the statement sequence (compound statement) like a single statement. If you put the sequence of statements inside a program construct such as an If statement, the statements become a statement sequence. For example:

```vbnet
If (divisor <> 0) Then
    result = dividend / divisor
Else
    lblOut.Text = errorMsg
    result = Integer.MaxValue
End If
```

If the divisor is zero, the computer both displays the error message and sets the value of result to Integer.MaxValue before continuing with whatever statement follows the If statement.

Statement sequences can be used in both branches of an If-Else statement. For example:

```vbnet
Dim errorMsg As String
Dim okMsg As String
errorMsg = "Division by zero is not allowed."
okMsg = "Division performed."
If (divisor <> 0) Then
    result = dividend / divisor
    lblOut.Text = okMsg
Else
    lblOut.Text = errorMsg
    result = Integer.MaxValue
End If
```

The If Form

Sometimes you run into a situation where you want to say, “If a certain condition exists, perform some action; otherwise, don’t do anything.” In other words, you want
the computer to skip a sequence of instructions if a certain condition isn’t met. You could do this by leaving the Else branch empty, using only the null statement:

```vbnet
If (a <= b) Then
    c = 20
Else

Better yet, you could simply leave off the Else part. The resulting statement is the If form of the If statement. This is its syntax template:

``IfStatement (the If form)

```vbnet
If (Expression) Then
    Statement
End If
```

Here’s an example of an If form.

```vbnet
If (age < 18) Then
    lblOut.Text = "Not an eligible "
End If
lblOut.Text = lblText.Out & "voter."
```

This statement means that if `age` is less than 18, first add the string "Not an eligible " to label `lblOut` and then add the string "voter." to the label. If `age` is not less than 18, skip the first statement and go directly to adding the string "voter." to the label. Figure 6.5 shows the flow of control for an If form.

![Figure 6.5 If flow of control](image-url)
Like the two branches in an *If-Else* form, the one branch in an *If* can be a block. For example, let’s say you are writing a program to compute income taxes. One of the lines on the tax form says, “Subtract line 23 from line 27 and enter result on line 24; if result is less than zero, enter zero and check box 24A.” You can use an *If* statement to do this in Visual Basic:

```vbnet
Dim checkMsg As String
checkMsg = "Check box 24A"
result = line17 - line23
If (result < 0R) Then
    lblOut.Text = checkMsg
    result = 0R
End If
line24 = result
```

This code does exactly what the tax form says it should. It computes the result of subtracting line 23 from line 17. Then it looks to see if `result` is less than zero. If it is, the fragment displays a message telling the user to check box 24A and then sets `result` to zero. Finally, the calculated result (or zero, if the result is less than zero) is stored into a variable named `line24`.

You can also write the *If* form without the `End If` at the end of the form. When you do this, the *Then-*clause of the *If* form is restricted to a single statement, which immediately follows the word *Then*. For example:

```vbnet
Dim checkMsg As String
checkMsg = "Check box 24A"
result = line17 - line23
If (result < 0.0) Then lblOut.Text = checkMsg
```

The statement that follows *Then* is executed if the assertion is true. It is the only statement that is executed based on the value of the assertion, however, because there is no statement sequence. Any statement that follows this line will, of course, be executed as part of the normal control flow of the program. If you want a statement sequence for the *Then-*clause, then you must use the keyword *Then* and put the statements below the clause.

**Nested If Statements**

There are no restrictions on what the statements in an *If* form can be. Therefore, an *If* within an *If* is okay. In fact, an *If* within an *If* within an *If* is legal. The only limitation here is that people cannot follow a structure that is too involved, and readability is one of the marks of a good program.

When we place an *If* within an *If*, we are creating a nested control structure. Control structures nest much like mixing bowls do, with smaller ones tucked inside larger ones. Here’s an example written in pseudocode:
In general, any problem that involves a *multiway branch* (more than two alternative courses of action) can be coded using nested *If* statements. For example, to store the name of a month into a string variable, we could use a sequence of *If* statements (unnested):

```plaintext
If (month = 1) Then
    monthName = "January"
End If
If (month = 2) Then
    monthName = "February"
End If
If (month = 3) Then
    monthName = "March"
End If
: 
If (month = 12) Then
    monthName = "December"
End If
```

But the equivalent nested *If* structure:

```plaintext
If (month = 1) Then
    monthName = "January"
Else
    If (month = 2) Then
        monthName = "February"
    Else
        If (month = 3) Then
            monthName = "March"
        Else
            If (month = 4)
```

...
is more efficient because it makes fewer comparisons. The first version—the sequence of independent _If_ statements—always tests every condition (all 12 of them), even if the first one is satisfied. In contrast, the nested _If_ solution skips all remaining comparisons after one alternative has been selected. As fast as modern computers are, many applications require so much computation that an inefficient algorithm can waste hours of computer time. Always be on the lookout for ways to make your programs more efficient, as long as doing so doesn’t make them difficult for other programmers to understand. It’s usually better to sacrifice a little efficiency for the sake of readability.

In the last example, notice how the indentation of the _Then- and _Else-clauses causes the statements to move continually to the right. Instead, we can use a special indentation style with deeply nested _If-Else_ statements to indicate that the complex structure is just choosing one of a set of alternatives. This general multiway branch is known as an _If-ElseIf_ control structure:

```c
If (month = 1) Then
    monthName = "January"
ElseIf (month = 2) Then
    monthName = "February"
ElseIf (month = 3) Then
    monthName = "March"
ElseIf (month = 4) Then
Else
    monthName = "December"
End If
```

Notice that _ElseIf_ is one word. This style prevents the indentation from marching continuously to the right. But, more importantly, it visually conveys the idea that we are using a 12-way branch based on the variable _month_.

Another particularly helpful use of the nested _If_ is when you want to select from a series of consecutive ranges of values. For example, suppose that you want to display a message indicating an appropriate activity for the outdoor temperature, given the following table.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swimming</td>
<td>Temperature &gt; 85</td>
</tr>
<tr>
<td>Tennis</td>
<td>70 &lt; temperature &lt;= 85</td>
</tr>
<tr>
<td>Golf</td>
<td>32 &lt; temperature &lt;= 70</td>
</tr>
<tr>
<td>Skiing</td>
<td>0 &lt; temperature &lt;= 32</td>
</tr>
<tr>
<td>Dancing</td>
<td>Temperature &lt;= 0</td>
</tr>
</tbody>
</table>
At first glance, you may be tempted to write a separate `If` statement for each range of temperatures. On closer examination, however, it is clear that these conditions are interdependent. That is, if one of the statements is executed, none of the others should be executed. We really are selecting one alternative from a set of possibilities—just the sort of situation in which we can use a nested `If` structure as a multiway branch. The only difference between this problem and our earlier example of printing the month name from its number is that we must check ranges of numbers in the `If` expressions of the branches.

When the ranges are consecutive, we can take advantage of that fact to make our code more efficient. We arrange the branches in consecutive order by range. Then, if a particular branch has been reached, we know that the preceding ranges have been eliminated from consideration. Thus, the `If` expressions must compare the temperature to only the lowest value of each range.

```basic
message = "The recommended activity is "
If (temperature > 85) Then
    message = message & "swimming."
ElseIf (temperature > 70) Then
    message = message & "tennis."
ElseIf (temperature > 32) Then
    message = message & "golf."
ElseIf (temperature > 0) Then
    message = message & "skiing."
Else
    message = message & "dancing."
End if
```

To see how this `If-ElseIf` structure works, consider the branch that tests for temperature greater than 70. If it has been reached, you know that temperature must be less than or equal to 85 because that condition causes this particular `ElseIf` branch to be taken. Thus, you need to test only whether temperature is above the bottom of this range (> 70). If that test fails, then you enter the next `ElseIf`-clause knowing that temperature must be less than or equal to 70. Each successive branch checks the bottom of its range until you reach the final `Else`, which takes care of all the remaining possibilities.

Note that if the ranges aren’t consecutive, then you must test the data value against both the highest and lowest value of each range. You still use an `If-ElseIf` structure because that is best for selecting a single branch from multiple possibilities, and you can arrange the ranges in consecutive order to make them easier for a human reader to follow. But there is no way to reduce the number of comparisons when there are gaps between the ranges.
Problem-Solving Case Study

A Simple Calculator

Problem  It’s useful to have a calculator that you can display on your computer screen when you look around and can’t find your hand-held one. Let’s write a Visual Basic application that simulates a simple calculator. Once we have the basic design it is easy to extend it to provide other functions.

Brainstorming  You pick up your pocket calculator. After all, this problem is one of simulating the pocket calculator. What are the objects that you see? A register on the top that shows what you are entering and displays the answer, an ON button, buttons for each of the ten decimal digits, buttons for each of the four basic operations, a CLEAR button, and a button marked with an equal sign. There are buttons for percent and square root, but you do not use them often enough to add them to the simulation. So the first list of proposed classes is as follows:

- register
- on button
- buttons for each digit
- buttons for four operations
- clear button
- equal button

Filtering  We certainly need a register to show the output. Do we need an ON button? No, running the program is the equivalent to the ON button. However, we do need some way of quitting the program. We could have a quit button or we could let the window closing end the program. We need an object to represent the calculator itself; that is, we need a form to hold all the other objects. What about the buttons for each digit? No, we can let the user input a number into a text field rather than pushing individual buttons. Do we need an ENTER button? No, we can let the button for the operation signal that the value is ready to be read. We do need buttons for each operation. We certainly need a CLEAR button to set the register value to zero. What about the equal button? On the calculator it signals that we want to see the results so far in the register. Instead of entering an equal button, let’s just display the result of each operation in the register as we go along.

Here is our second pass at the classes for our application:
The register is simply a label that displays what is put into it, so it actually has no further responsibility. The form object is a container into which labels, text boxes, and buttons are placed, so it collaborates with each of these. The text box for the input has no responsibilities; it just collaborates with the appropriate button handler. The label for the text box has only one responsibility: to display the input prompt. What happens when the operation buttons are pressed? The four operator buttons are event sources; they have no responsibility, but their handlers do. When one is pressed, Visual Basic calls the appropriate button handler, which collaborates with the text box to input a value, performs a calculation, and send the value to the register. The CLEAR button is different in function from the others but its only responsibility is to be an event source. However, its event handler must collaborate with the register to set it to zero. It makes sense to use one button handler for the four arithmetic buttons, but a separate one for the CLEAR button.

The application has responsibility for setting up the window, handling the button events through the inclusion of the two event subprocedures, and handling the window-closing event.

CRC Cards All of the objects are instances of classes provided in the Visual Basic Winforms namespace. So the only CRC card is for class Form.

<table>
<thead>
<tr>
<th>Class Name: Calculator</th>
<th>Superclass:</th>
<th>Subclasses:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
<td>Collaborations</td>
<td>Subclasses:</td>
</tr>
<tr>
<td>Prepare the window for input</td>
<td>Textbox, Label, String, Buttons</td>
<td></td>
</tr>
<tr>
<td>Handle numeric button event</td>
<td>Buttons, register (textbox)</td>
<td></td>
</tr>
<tr>
<td>Handle clear button event</td>
<td>Buttons, register (textbox)</td>
<td></td>
</tr>
<tr>
<td>Handle window closing</td>
<td>Frame</td>
<td></td>
</tr>
</tbody>
</table>

Responsibility Algorithms Do you remember in Chapter 1, after the long first program, that we said that much of the code dealt with input and output, which would soon become second nature to you? Well the first responsibility should sound comfortingly familiar. It is exactly the same as the first responsibility for Chapter 5’s Case Study. Does this mean that we can just cut
and paste the algorithms? No, the objects in the window are not the same, but the steps used to create them are the same, so we can reuse the top-level module.

Prepare Form for Input  

**Level 1**

Place controls on form
Set properties for controls, including names and any text

The first step in creating a user interface is to place the controls that are going to make up the interface on the form. The form will include two labels (one for input and one for the register), a text box for entering a data value, four buttons for the user to click to indicate the operation when a value is ready for input, and a button for the user to click to clear the register.

Before we go on, let's simplify the problem somewhat. Let's first create a calculator with only plus and minus functions. After we verify this simplified version of the program, we can come back and add division and multiplication (see Case Study Follow-Up Question 1). We also set the TextAlign property of the register and the text box to Right to make the numbers look better in the calculator.

Set Control Properties  

**Level 2**

Set Form1's Text property to “Calculator”
Name one label lblResult, set its Text property to “Result”
Name one label lblRegister, set its Text property to “0.0”
Name text box txtInput, set its Text property to “”
Name one button btnAdd, set its Text property to “+”
Name one button btnSubtract, set its Text property to “-”
Name one button btnClear, set its Text property to “Clear”
Set txtInput TextAlign property to Right
Set lblRegister TextAlign property to Right

The form, in design view, should look like this:

![Calculator Form Design](image)

We have now completed the first responsibility for the Calculator program. The second responsibility is to handle the numeric buttons; that is, write the Click event code for the buttons. What should the Click event do when an add or subtract button is pushed? First, it should get the value to be used in the calculation from the text box and convert it to a real number. If the user
enters a string that does not represent a legitimate real number, a FormatException may be thrown. In Chapter 10 we show you how to handle this exception; here we assume correct input.

The next step is to determine how the arithmetic operations (addition and subtraction) should be performed. Because we only have one text box to store a number, we have to move each entry in the text box to a variable. This simply requires setting a variable, result, to 0, adding or subtracting the value in the text box to the value in result, and then clearing the text box so another number can be entered. Finally, we need to put the focus back in the text box so that the user can enter another value without having to move the cursor to the text box.

---

**Handle Addition Button Click Event**  
Level 1

- Get the entry string from text box
- Set operand to numerical value (Double) of entry string
- Set result to result + operand
- Display result in register
- Clear text box
- Set focus in text box

**Handle Subtraction Button Click Event**  
Level 1

- Get the entry string from text box
- Set operand to numerical value (Double) of entry string
- Set result to result — operand
- Display result in register
- Clear text box
- Set focus in text box

The third responsibility is even simpler; all we have to do is set the register and text box to zero as well as the variable that contains the same value, and put the focus back in the text box. These actions are performed in the Clear button’s Click event.

**Handle Clear Button Click Event**  
Level 1

- Set result to zero
- Display result in register
- Clear text box
- Set focus

The fourth responsibility, handling window closing, is taken care of automatically by Visual Basic.
The coding of the program into Visual Basic is now easy and straightforward. Here is the result.

' Import user interface classes
Imports System.ComponentModel
Imports System.Drawing
Imports System.WinForms

Public Class Form1
  Inherits System.WinForms.Form

  Dim result As Double ' Holds current value
  Dim operand As Double ' Holds input value

  Public Sub New()
    MyBase.New()
    Form1 = Me
    InitializeComponent()
    result = 0R ' Initialize result to 0.0
  End Sub

  ' Form overrides dispose to clean up the component list.
  Public Overrides Sub Dispose()
    MyBase.Dispose() ' Close the frame
    components.Dispose() ' Exit the program
  End Sub

  #Region " Windows Form Designer generated code "
  #End Region

  Protected Sub btnSubtraction_Click(ByVal sender As Object, ByVal e As System.EventArgs)
  End Sub

  Objects and Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form1</td>
<td>Form</td>
<td>Form to display on screen</td>
</tr>
<tr>
<td>lblResultText</td>
<td>Label</td>
<td>Indicates output area</td>
</tr>
<tr>
<td>lblResult</td>
<td>Label</td>
<td>Label to show output</td>
</tr>
<tr>
<td>lblEntry</td>
<td>Label</td>
<td>Label indicating input text box</td>
</tr>
<tr>
<td>btnAddition</td>
<td>Button</td>
<td>Addition button</td>
</tr>
<tr>
<td>btnSubtraction</td>
<td>Button</td>
<td>Subtraction button</td>
</tr>
<tr>
<td>btnClear</td>
<td>Button</td>
<td>Clear button</td>
</tr>
<tr>
<td>result</td>
<td>Double</td>
<td>Holds current value</td>
</tr>
<tr>
<td>operand</td>
<td>Double</td>
<td>Holds input value</td>
</tr>
</tbody>
</table>


When we run the program, it displays a window like the one shown below.

Testing and Debugging

In Chapter 1, we discussed the problem-solving and implementation phases of computer programming. Testing is an integral part of both phases. Testing in the problem-solving phase is done after the solution is developed but before it is implemented. In the implementation phase, we test after the algorithm is translated into a program, and again
after the program has compiled successfully. The compilation itself constitutes another stage of testing that is performed automatically.

**Testing Strategies**

When an individual programmer is designing and implementing a program, he or she can find many software errors with pencil and paper. **Desk checking** the design solution is a very common method of manually verifying a design or program. The programmer writes down essential data (variables, input values, parameters of subprograms, and so on) and walks through the design, manually simulating the actions of the computer and recording changes in the data on paper. Portions of the design or code that are complex or that are a source of concern should be double-checked.

Desk checking can be done by an individual, but most sizable computer programs are developed by teams of programmers. Two extensions of desk checking that are used effectively by programming teams are design or code **walk-throughs** and **inspections**. These are formal team activities, the intention of which is to move the responsibility for uncovering bugs from the individual programmer to the group. Because testing is time-consuming and errors cost more the later they are discovered, the goal is to identify errors before testing begins.

In a walk-through the team performs a manual simulation of the design or program with sample test inputs, keeping track of the program’s data by hand. Unlike thorough program testing, the walk-through is not intended to simulate all possible test cases. Instead, its purpose is to stimulate discussion about the way the programmer chose to design or implement the program’s requirements.

At an inspection, a reader (not necessarily the program’s author) goes through the design or code line by line. Inspection participants point out errors, which are recorded on an inspection report. Some errors are uncovered just by the process of reading aloud. Others may have been noted by team members during their pre-inspection preparation. As with the walk-through, the chief benefit of the team meeting is the discussion that takes place among team members. This interaction among programmers, testers, and other team members can uncover many program errors long before the testing stage begins.

At the high-level design stage, the design should be compared to the application requirements to make sure that all required responsibilities have been included and that this application or class correctly interfaces with other software in the system. At the low-level design stage, when the design has been filled out with more details, it should be reinspected before it is implemented.

After the code is written, you should go over it line by line to be sure that you’ve faithfully reproduced the algorithm in a code walk-through. In a team programming situation, you ask other team members to walk through the algorithm and code with you, to double-check the design and code.
You also should take some actual values and hand-calculate what the output should be by doing an execution trace (or hand trace). When the program is executed, you can use these same values as input and then check the results. The computer is a very literal device—it does exactly what we tell it to do, which may or may not be what we want it to do. We try to make sure that a program does what we want by tracing the execution of the statements.

When a program contains branches, it’s a good idea to retrace its execution with different input data so that each branch is traced at least once. In the next section, we describe how to develop data sets that test each of a program’s branches.

To test an application with functions, we need to execute each function. If a function contains branches, we need to execute each branch at least once and verify the results. For example, in the Problem-Solving Case Study, there are three Click methods (a method is similar to a function). Therefore, we need at least three data sets to test the methods. The following combination of buttons and input values cause all the methods to be executed:

<table>
<thead>
<tr>
<th>Button</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>+ 12.5</td>
</tr>
<tr>
<td>Set 2</td>
<td>− 2.5</td>
</tr>
<tr>
<td>Set 3</td>
<td>Clear</td>
</tr>
</tbody>
</table>

Every event in the program is executed at least once through this series of test runs, which provide what is called minimum complete coverage of the program’s branching structure. Whenever you test a program with branches in it, you should design a series of tests that covers all of the branches. Because an action in one branch of a program often affects processing in a later branch, it is critical to test as many combinations of branches, or paths, through a program as possible. By doing so, you can be sure there are no interdependencies that could cause problems. Should you try all possible paths? Yes, in theory you should. However, the number of paths in even a small program can be very large.

The approach to testing that we’ve used here is called code coverage because the test data is designed by looking at the code of the program. Code coverage is also called white-box (or clear-box) testing because you are allowed to see the program code while designing the tests. Another approach to testing, data coverage, attempts to test as many allowable data values as possible without regard to the program code. Because you need not see the code, it is also called black-box testing—the same set of tests would be used even if the code were hidden in a black box. Complete data coverage is as impractical as complete code coverage for many programs. For example, the btnAddition_Click event reads one Double value and thus has an immense number of possible input values.

Often, testing is a combination of these two strategies. Instead of trying every possible data value (data coverage), the idea is to examine the code (code coverage) and look for ranges of values for which processing is identical. Then one tests the values at
the boundaries and, sometimes, a value in the middle of each range. For example, a simple condition such as

\[ \alpha < 0 \]

divides the integers into two ranges:

1. \( \text{Integer.MinValue} \) through \(-1\)
2. \( 0 \) through \( \text{Integer.MaxValue} \)

Thus, you should test the four values \( \text{Integer.MinValue} \), \(-1\), \(0\), and \( \text{Integer.MaxValue} \). A compound condition such as

\[ \alpha \geq 0 \text{ And } \alpha \leq 100 \]

divides the integers into three ranges:

1. \( \text{Integer.MinValue} \) through \(-1\)
2. \( 0 \) through \( 100 \)
3. \( 101 \) through \( \text{Integer.MaxValue} \)

Thus, you have six values to test. In addition, to verify that the relational operators are correct, you should test for values of \(1 (> 0)\) and \(99 (< 100)\).

Conditional branches are only one factor in developing a testing strategy. We consider more of these factors in later chapters.

The Test Plan

We’ve discussed strategies and techniques for testing programs, but how do you approach the testing of a specific program? You do it by designing and implementing a test plan—a document that specifies the test cases that should be tried, the reason for each test case, and the expected output. Test plan implementation involves running the program using the data specified by the test cases in the plan and checking and recording the results.

The test plan should be developed together with the design. The following table shows a partial test plan for the Calculator application. The first test case assumes that the calculator has an initial value of \(0.0\). Successive test cases use the preceding result as one of the two operands for a calculation. To be more thorough, we should check that the program properly handles closing the window, and we should include some negative as well as positive values.

Implementing a test plan does not guarantee that a program is completely correct. It means only that a careful, systematic test of the program has not demonstrated any
bugs. The situation shown in Figure 6.6 is analogous to trying to test a program without a plan – depending only on luck, you may completely miss the fact that a program contains numerous errors. Developing and implementing a written test plan, on the other hand, casts a wide net that is much more likely to find errors.

**Test Plan for the Calculator Application**

<table>
<thead>
<tr>
<th>Reason for Test Case</th>
<th>Input Values</th>
<th>Expected Output</th>
<th>Observed Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test addition command</td>
<td>12.5, +</td>
<td>12.5</td>
<td></td>
</tr>
<tr>
<td>Test subtraction command</td>
<td>2.5, -</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Test clear command</td>
<td>none, Clear</td>
<td>0.0</td>
<td></td>
</tr>
<tr>
<td>Test window closing</td>
<td>none</td>
<td>window disappears</td>
<td></td>
</tr>
</tbody>
</table>

**Tests Performed Automatically during Compilation and Execution**

Once a program is coded and test data has been prepared, it is ready for compiling. The compiler has two responsibilities: to report any errors and (if there are no errors) to translate the program into object code or Bytecode.

Errors can be syntactic or semantic. The compiler finds syntactic errors. For example, the compiler warns you when reserved words are misspelled, identifiers are undeclared, and operand types are mismatched. These errors are marked even before the
program is compiled. When the compiler finds an error while you are entering source code, it will underline the error with a blue squiggly line. This gives you a visual clue that you have an error, and you must fix the error (and any others) before the program will run. The compiler won’t find all of your typing errors. If you type \( > \) instead of \( < \), you won’t get an error message; instead, you get erroneous results when you test the program. It’s up to you to design a test plan and carefully check the code to detect errors of this type.

Semantic errors (also called *logic errors*) are mistakes that give you the wrong answer. They are more difficult to locate than syntactic errors and usually surface when a program is executing. Visual Basic detects only the most obvious semantic errors—those that result in an invalid operation (integer division by zero, for example). Although typing errors sometimes cause semantic errors, they are more often a product of a faulty algorithm design.

By walking through the algorithm and the code, tracing the execution of the program, and developing a thorough test strategy, you should be able to avoid, or at least quickly locate, semantic errors in your programs.

Figure 6.7 illustrates the testing process we’ve been discussing. The figure shows where syntax and semantic errors occur and in which phase they can be corrected.

**Testing and Debugging Hints**

1. The equal sign (=) can be used for assignment and for testing equality. Be careful how you use it.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Result</th>
<th>Testing Technique</th>
<th>Type of Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem solving</td>
<td>Algorithm</td>
<td>Algorithm walk-through</td>
<td>Semantic</td>
</tr>
<tr>
<td>Implementation</td>
<td>Coded program</td>
<td>Code walk-through, Trace</td>
<td>Syntax</td>
</tr>
<tr>
<td>Compilation</td>
<td>Object program</td>
<td>Compiler-generated error messages</td>
<td>Syntax</td>
</tr>
<tr>
<td>Execution</td>
<td>Output</td>
<td>Implement test plan</td>
<td>Typographical semantic</td>
</tr>
</tbody>
</table>

*Figure 6.7  Testing process*
2. If you use extra parentheses for clarity, be sure that the opening and closing parentheses match up. To verify that parentheses are properly paired, start with the innermost pair and draw a line connecting them. Do the same for the others, working your way out to the outermost pair. For example,

```
If( ( (total/scores) > 50) And ( (total/(scores - 1) ) < 100) ) Then
```

3. Here is a quick way to tell whether you have an equal number of opening and closing parentheses. The scheme uses a single number (the “magic number”), whose value initially is 0. Scan the expression from left to right. At each opening parenthesis, add 1 to the magic number; at each closing parenthesis, subtract 1. At the final closing parenthesis, the magic number should be 0. In the following example the digits indicate the total number so far, working from left to right.

```
If   ( (total/scores) > 50) And ( (total/(scores–1) ) < 100) ) Then
0    1 2 3            2     1     2 3      4        3 2      1 0
```

4. Don’t use =< to mean “less than or equal to”; only the symbol <= works. Likewise, => is invalid for “greater than or equal to”; you must use => for this operation.

5. Don’t compare strings with the = operator. Use the associated instance methods such as Equal and CompareTo. When testing for alphabetical order, remember to convert the strings to the same case before comparison.

6. When comparing values of different types, use type conversion functions to convert the values before comparison.

7. Don’t compare floating-point types for exact equality. Check that the difference between the values is less than some small amount.

8. Test for bad data. If a data value must be positive, use an If statement to test the value. If the value is negative or 0, an error message should be displayed; otherwise, processing should continue.

9. Take some sample values and try them by hand. Develop a test plan before you start testing your program.

10. If your program produces an answer that does not agree with a value you’ve calculated by hand, try these suggestions:

    • Redo your arithmetic.
    • Recheck your input data.
    • Carefully go over the section of code that does the calculation. If you’re in doubt about the order in which the operations are performed, insert clarifying parentheses.
• Check for integer overflow. The value of an Integer variable may have exceeded Integer.MaxValue in the middle of a calculation. Visual Basic doesn’t give an error message when this happens.
• Check the conditions in branching statements to be sure that the correct branch is taken under all circumstances.

Summary of Classes

In this chapter we introduced additional instance methods for the String class that enable us to compare the values contained in strings.

Namespace Name

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td></td>
</tr>
<tr>
<td><strong>Value-Returning Instance Methods:</strong></td>
<td></td>
</tr>
<tr>
<td>Equals(String)</td>
<td>True if string contents match parameter</td>
</tr>
<tr>
<td>CompareTo(String)</td>
<td>0 if equal, positive Integer if parameter comes before string, negative Integer if parameter comes after string</td>
</tr>
<tr>
<td><strong>ToString</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ToLower</strong></td>
<td>Returns an identical string, with all characters lowercase</td>
</tr>
<tr>
<td><strong>ToUpper</strong></td>
<td>Returns an identical string, with all characters uppercase</td>
</tr>
</tbody>
</table>

Summary

Using logical expressions is a way of asking questions while a program is running. The program evaluates each logical expression, producing the value True if the expression is true or the value False if the expression is not true.

The If statement allows you to take different paths through a program based on the value of a logical expression. The If-Else statement is used to choose between two courses of action; the If statement is used to choose whether or not to take a particular course of action. The branches of an If or If-Else can be any statement, simple or compound. They can even be another If statement.

The algorithm walk-through is a manual simulation of the algorithm at the design phase. By testing our design in the problem-solving phase, we can eliminate errors that can be more difficult to detect in the implementation phase.

An execution trace is a way of finding program errors once we’ve entered the implementation phase. It’s a good idea to trace a program before you run it, so that you have
some sample results against which to check the program’s output. A written test plan is an essential part of any program development effort.

**Quick Check**

1. Write a Visual Basic expression that compares the variable `letter` to the constant "Z" and yields `True` if `letter` is less than "Z". (pp. 217–218)
2. Write a Visual Basic expression that yields `True` if `letter` is between "A" and "Z" inclusive. (pp. 217–218)
3. What form of the `If` statement would you use to make a Visual Basic program print out “Is an uppercase letter” if the value in `letter` is between "A" and "Z" inclusive, and print out “Is not an uppercase letter” if the value in `letter` is outside that range? (pp. 227–228)
4. What form of the `If` statement would you use to make a Visual Basic program print out “Is a digit” only if the value in the variable `someChar` is between "0" and "9" inclusive? (pp. 229–230)
5. On the telephone, each of the digits 2 through 9 has a segment of the alphabet associated with it. What kind of control structure would you use to decide which segment a given letter falls into and to print out the corresponding digit? (pp. 231–232)
6. If operands of a relational expression are characters rather than numeric values, what do the relational operators measure?
7. What is the event that is triggered when a user pushes a button on a form? (pp. 237–239)
8. In what phase of the program development process should you carry out an execution trace? (pp. 241–242)
9. You’ve written an application that displays the corresponding digit on a phone, given a letter of the alphabet. Everything seems to work right except that you can’t get the digit “5” to display; you keep getting the digit "6". What steps would you take to find and fix this bug? (pp. 243–245)

**Answers**

1. `letter < "Z"c`  
2. `letter >= "A"c And letter <= "Z"c`  
3. The `If-Else` form  
4. The `If` form  
5. A nested `If` statement  
6. The relative position of the characters within the collating sequence of the character set  
7. Click  
8. The implementation phase  
9. Carefully review the section of code that should print out "5". Check the branching condition and the output statement there. Try some sample values by hand.

**Exam Preparation Exercises**

1. What is the purpose of a control structure?
2. What is a logical expression?
3. Given the following relational expressions, state in English what they say.
Expression | Meaning in English
--- | ---
one = two
one <> two
one > two
one < two
one >= two
one <= two

4. Given the values for the Boolean variables \( x, y, \) and \( z \):

\[
x = \text{True}, \quad y = \text{False}, \quad z = \text{True}
\]

evaluate the following logical expressions. In the blank next to each expression, write a T if the result is True or an F if the result is False.

_____ a. \( x \) And \( y \) Or \( x \) And \( z \)
_____ b. \( (x \ Or \ Not(y)) \ And \ (Not(x) \ Or \ z) \)
_____ c. \( x \ Or \ y \ And \ z \)
_____ d. \( Not(x \ Or \ y) \ And \ z \)

5. Given these values for variables \( i, j, p, \) and \( q \):

\[
i = 10, \quad j = 19, \quad p = \text{True}, \quad q = \text{False}
\]

add parentheses (if necessary) to the expressions below so that they evaluate to True.

_____ a. \( i = j \) Or \( p \)
_____ b. \( i >= j \) Or \( i <= j \ And \ p \)
_____ c. \( \text{Not}(p) \) Or \( p \)
_____ d. \( \text{Not}(q) \) And \( q \)

6. Given these values for the Integer variables \( j, k, m, \) and \( n \):

\[
j = 6, \quad k = 7, \quad m = 11, \quad n = 11
\]

What is the value of \( s \) in the following code?

\[
s = "\text{Madam}\"
\]
If \( (j < k) \) Then
    If \( (m <> n) \) Then
        \( s = s \) & "How"
    Else
        \( s = s \) & "Now"
    End If
End If
If \( (j >= m) \) Then
    \( s = s \) & "Cow"
Else
    \( s = s \) & "Adam"
End If
7. Given the Integer variables \( x, y, \) and \( z \), where \( x \) contains 3, \( y \) contains 7, and \( z \) contains 6, what is the value of \( s \) in each of the following code fragments?

a. If \( (x \leq 3) \) Then 
   \[ s = "x" \& "y" \]
   End If

b. If \( (x \neq -1) \) Then 
   \[ s = "The value of x is " \& x \]
   Else 
   \[ s = "The value of y is " \& y \]
   End If

c. If \( (x \neq -1) \) Then 
   \[ s = x \]
   \[ s = y \]
   \[ s = z \]
   Else 
   \[ s = "y" \]
   \[ s = "z" \]
   End If

8. Given this code fragment:

\[
\begin{align*}
& \text{If (height} \geq \text{minHeight) Then} \\
& \quad \text{If (weight} \geq \text{minWeight) Then} \\
& \quad \quad \text{Messagebox.Show("Eligible to serve."\)}} \\
& \quad \text{Else} \\
& \quad \quad \text{Messagebox.Show("Too light to serve."\)}} \\
& \quad \text{End If} \\
& \text{Else} \\
& \quad \text{If (weight} \geq \text{minWeight) Then} \\
& \quad \quad \text{Messagebox.Show("Too short to serve."\)}} \\
& \quad \text{Else} \\
& \quad \quad \text{Messagebox.Show("Too short and too light to serve."\)}} \\
& \text{End If} \\
& \text{End If}
\end{align*}
\]

9. Match each logical expression in the left column with the logical expression in the right column that tests for the same condition.

\[
\begin{align*}
\text{_____} & \quad \text{a.}\ x < y \text{ And } y < z & \quad (1) \quad \text{Not}(x \neq y) \text{ And } y = z \\
\text{_____} & \quad \text{b.}\ x > y \text{ And } y \geq z & \quad (2) \quad \text{Not}(x < y \text{ Or } y < z) \\
\text{_____} & \quad \text{c.}\ x \neq y \text{ Or } y = z & \quad (3) \quad (y < z \text{ Or } y = z) \text{ Or } x = y \\
\text{_____} & \quad \text{d.}\ x = y \text{ Or } y \leq z & \quad (4) \quad \text{Not}(x \geq y) \text{ And } \text{Not}(y \leq z) \\
\text{_____} & \quad \text{e.}\ x = y \text{ And } y = z & \quad (5) \quad \text{Not}(x = y \text{ And } y \neq z)
\end{align*}
\]

10. The following expressions make sense but are invalid according to Visual Basic’s rules of syntax. Rewrite them so that they are valid logical expressions. (All the variables are of type Integer.)

a. \( x < y \leq z \)

b. \( x, y, \) and \( z \) are greater than 0
c. \( x \) is equal to neither \( y \) nor \( z \)

d. \( x \) is equal to \( y \) and \( z \)

11. Given these values for the Boolean variables \( x, y, \) and \( z \):

\[
x = \text{True}, \ y = \text{True}, \ z = \text{False}
\]

Indicate whether each expression is True (T) or False (F).

_____ a. \( \neg(y \lor z) \lor x \)

_____ b. \( z \land x \land y \)

_____ c. \( \neg y \lor (z \lor \neg(x)) \)

_____ d. \( z \lor (x \land (y \lor z)) \)

_____ e. \( x \lor x \land z \)

12. For each of the following problems, decide which is more appropriate, an *If-Else* or an *If*. Explain your answers.

a. Students who are candidates for admission to a college submit their SAT scores. If a student’s score is equal to or above a certain value, print a letter of acceptance for the student. Otherwise, print a rejection notice.

b. For employees who work more than 40 hours a week, calculate overtime pay and add it to their regular pay.

c. In solving a quadratic equation, whenever the value of the discriminant (the quantity under the square root sign) is negative, print out a message noting that the roots are complex (imaginary) numbers.

d. In a computer-controlled sawmill, if a cross section of a log is greater than certain dimensions, adjust the saw to cut four-inch by eight-inch beams; otherwise, adjust the saw to cut two-inch by four-inch studs.

13. What causes the error message “ Else Without If” when this code is written?

```vbnet
If (mileage < 24R) Then
    lblOut.Text = "Gas"
End If
Else
    lblOut.Text = "Fuel efficient"
```

14. The following code fragment is supposed to print “Type AB” when Boolean variable \( \text{typeA} \) and \( \text{typeB} \) are both \text{True}, and print “Type O” when both variables are \text{False}. Instead, it prints “Type O” whenever just one of the variables is \text{False}. Rewrite the code to make the code segment work the way it should.

```vbnet
If (typeA Or typeB) Then
    If (typeA And typeB) Then msgbox("Type AB")
Else
    msgbox("Type O")
End If
```
15. The nested *If* structure below has five possible branches depending on the values read into *Char* variables *ch1*, *ch2*, and *ch3*. To test the structure, you need five sets of data, each set using a different branch. Create the five data sets.

```
If (ch1 = ch2) Then
    If (ch2 = ch3) Then
        MessageBox.Show("All initials are the same.")
    Else
        MessageBox.Show("First two are the same.")
    End If
ElseIf (ch2 = ch3) Then
    MessageBox.Show("Last two are the same.")
ElseIf (ch1 = ch3) Then
    MessageBox.Show("First and last are the same.")
Else
    MessageBox.Show("All initials are the same.")
End If
```

a. Test data set 1: `ch1 = _____` `ch2 = _____` `ch3 = _____`
b. Test data set 2: `ch1 = _____` `ch2 = _____` `ch3 = _____`
c. Test data set 3: `ch1 = _____` `ch2 = _____` `ch3 = _____`
d. Test data set 4: `ch1 = _____` `ch2 = _____` `ch3 = _____`
e. Test data set 5: `ch1 = _____` `ch2 = _____` `ch3 = _____`

16. If *x* and *y* are *Boolean* variables, do the following two expressions test the same condition?

```
x <> y
(x Or y) And Not(x And y)
```

17. The following *If* condition is made up of three relational expressions:

```
If (i >= 10 And i <= 20 And i <> 16) Then
    j = 4
End If
```

If *i* contains the value 25 when this *If* statement is executed, which relational expression(s) does the computer evaluate? (Remember that Visual Basic uses short-circuit evaluation.)

18. a. If strings cannot be compared using the relational operators in Visual Basic, how can you compare two strings?
b. Fill in the following table that describes methods that can be applied to string objects.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Parameter</th>
<th>Returns</th>
<th>English Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CompareTo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToUpper</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToLower</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
19. What is the name of the event that is called when a button is pushed?

Programming Warm-Up Exercises

1. Declare eligible to be a Boolean variable, and assign it the value True.

2. Write a statement that sets the Boolean variable available to True if numberOrdered is less than or equal to numberOnHand minus numberReserved.

3. Write a statement containing a logical expression that assigns True to the Boolean variable isCandidate if satScore is greater than or equal to 1100, gpa is not less than 2.5, and age is greater than 15. Otherwise, isCandidate should be False.

4. Given the declarations
   Dim leftPage As Boolean
   Dim pageNumber As Integer

   write a statement that sets leftPage to True if pageNumber is even. (Hint: Consider what the remainders are when you divide different integers by two.)

5. Write an If statement (or a series of If statements) that assigns to the variable biggest the greatest value contained in variables m, n, and o. Assume the three values are distinct.

6. Rewrite the following sequence of If statements as a single If-Else statement.

   If (year Mod 4 = 0) Then
       Messagebox.Show(year & " is a leap year.")
   End If
   If (year Mod 4 <> 0) Then
       year = year + 4 - year Mod 4
       Messagebox.Show(year & " is the next leap year.")
   End If

7. Simplify the following program segment, taking out unnecessary comparisons. Assume that age is an Integer variable.

   If (age > 64) Then
       Messagebox.Show("Senior voter")
   End If
   If (age < 18) Then
       Messagebox.Show("Under age")
   End If
   If (age >= 18 And age < 65) Then
       Messagebox.Show("Regular Voter")
   End If

8. The following program fragment is supposed to print out the values 25, 60, and 8, in that order. Instead, it prints out 50, 60, and 4. Why?
Chapter 6: Conditions, Logical Expressions, and Selection Control Structures

length = 25
width = 60
If (length = 50) Then
    height = 4
Else
    height = 8
End If
Messagebox.Show(length & " " & width & " " & height)

9. The following Visual Basic program segment is almost unreadable because of the lack of indentation. Fix the indentation to make the program more readable.

' This is a nonsense program segment
If (a > 0) Then
    If (a < 20) Then
        Messagebox.Show("A is in range.")
        b = 5
    Else
        Messagebox.Show("A is too large.")
    End If
b = 3
Else
    Messagebox.Show("A is too small.")
    Messagebox.Show("All done.")
End If

10. Given the Single variables x1, x2, y1, y2, and m, write a program segment to find the slope of a line through the two points (x1, y1) and (x2, y2). Use the formula:

\[ m = \frac{y_1 - y_2}{x_1 - x_2} \]

to determine the slope of the line. If x1 equals x2, the line is vertical and the slope is undefined. The segment should write the slope with an appropriate label. If the slope is undefined, it should write the message “Slope undefined.”

11. Given the Single variables a, b, c, root1, root2, and discriminant, write a program segment to determine whether the roots of a quadratic polynomial are real or complex (imaginary). If the roots are real, find them and assign them to root1 and root2. If they are complex, write the message “No real roots.”

The formula for the solution to the quadratic equation is

\[ \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]


The ± means “plus or minus” and indicates that there are two solutions to the equation: one in which the result of the square root is added to $-b$ and one in which the result is subtracted from $-b$. The roots are real if the discriminant (the quantity under the square root sign) is not negative.

12. Provide a user interface with two buttons, Enter and Quit.
   a. Add the buttons to a form.
   b. Name the buttons.
   c. Write the statements that access a button’s name from within the Click event.
   d. Write the statements that display each button’s name on the form when the button is pushed.

Programming Problems

1. Design and write a Visual Basic application that inputs a single letter and prints out the corresponding digit on the telephone. The letters and digits on a telephone are grouped this way:

   \begin{align*}
   2 &= \text{ABC} & 4 &= \text{GHI} & 6 &= \text{MNO} & 8 &= \text{TUV} \\
   3 &= \text{DEF} & 5 &= \text{JKL} & 7 &= \text{PRS} & 9 &= \text{WXYZ}
   \end{align*}

   No digit corresponds to either Q or Z. For these two letters, your application should print a message indicating that they are not used on a telephone.

   The application should have two buttons, one to enter input and one to quit. The screen dialog might look like this:

   \begin{itemize}
   \item Enter a single letter, and I will tell you what the corresponding digit is on the telephone.
   \item \textbf{R}
   \end{itemize}

   The digit 7 corresponds to the letter \textbf{R} on the telephone.

   Here’s another example:

   \begin{itemize}
   \item Enter a single letter, and I will tell you what the corresponding digit is on the telephone.
   \item \textbf{Q}
   \end{itemize}

   There is no digit on the telephone that corresponds to \textbf{Q}.

   Your program should print a message indicating that there is no matching digit for any nonalphabetic character the user enters. Also, the program should recognize only uppercase letters. Include the lowercase letters with the invalid characters. Prompt the user with an informative message for the input value, as shown above.
Chapter 6: Conditions, Logical Expressions, and Selection Control Structures

Use proper indentation, appropriate comments, and meaningful identifiers throughout the program.

2. People who deal with historical dates use a number called the Julian day to calculate the number of days between two events. The Julian day is the number of days that have elapsed since January 1, 4713 B.C. For example, the Julian day for October 16, 1956 is 2,435,763. There are formulas for computing the Julian Day from a given date and vice versa.

One very simple formula computes the day of the week from a given Julian day:

\[
\text{Day of the week} = (\text{Julian day} + 1) \mod 7
\]

where Mod is the Visual Basic modulus operator. This formula gives a result of 0 for Sunday, 1 for Monday, and so on up to 6 for Saturday. For Julian day 2,435,763, the result is 2 (a Tuesday). Your job is to write a Visual Basic application that requests and prints out the name of the day that corresponds to that number. Use two buttons, Done and Quit. The application continues until the Quit button is pushed.

Your screen dialog might look like this:

Enter a Julian day number and press Done. Press Quit to quit.
2451545
Julian day number 2451545 is a Saturday.
2451547
Julian day number 2451547 is a Monday.

3. You can compute the date for any Easter Sunday from 1982 to 2048 as follows (all variables are of type Integer):

\[
\begin{align*}
a & \text{ is year} \mod 19 \\
b & \text{ is year} \mod 4 \\
c & \text{ is year} \mod 7 \\
d & (19 \times a + 24) \mod 30 \\
e & (2 \times b + 4 \times c + 6 \times d + 5) \mod 7 \\
\end{align*}
\]

Easter Sunday is March \((22 + d + e)\).\(^3\)

For example, Easter Sunday in 1985 is April 7.

Write an application that inputs the year and outputs the date (month and day) of Easter Sunday for that year. Use two buttons.

4. The algorithm for computing the date of Easter can be extended easily to work with any year from 1900 to 2099. There are four years—1954, 1981, 2049, and 2076—for which the algorithm gives a date that is seven days later than it should be. Modify the application for Exercise 3 to check for these years and subtract 7 from the day of the month. This correction does not cause the month to change.

\[^3\text{Notice that this formula can give a date in April.}\]
Be sure to change the documentation for the program to reflect its broadened capabilities.

5. Write a Visual Basic application that calculates and prints the diameter, the circumference, or the area of a circle, given the radius. The application should have three buttons, one for diameter, one for circumference, and one for area. The user should be prompted to enter the radius in floating-point form and press the appropriate button. The output should be labeled appropriately. For example, if the input is 6.75 and the area button is pressed, your program should print something like this:

The area of a circle with radius 6.75 is 143.14

Here are the formulas you need:

\[
\begin{align*}
\text{Diameter} &= 2r \\
\text{Circumference} &= 2\pi r \\
\text{Area of a circle} &= \pi r^2
\end{align*}
\]

where \( r \) is the radius. Use 3.14159265 for \( \pi \).

6. The factorial of a number \( n \) is \( n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1 \). Stirling’s formula approximates the factorial for large values of \( n \):

\[
\text{Factorial of } n = \frac{n^n \sqrt{2\pi n}}{e^n}
\]

where \( \pi = 3.14159265 \) and \( e = 2.718282 \).

Write a Visual Basic program that inputs an integer value (that stores it into a \texttt{Double} variable \( n \)), calculates the factorial of \( n \) using Stirling’s formula, assigns the (rounded) result to a \texttt{Long} integer variable, and then prints the result appropriately labeled. Depending on the value of \( n \), you should obtain one of these results:

- A numerical result.
- If \( n \) equals 0, the factorial is defined to be 1.
- If \( n \) is less than 0, the factorial is undefined.
- If \( n \) is too large, the result exceeds \texttt{Long.MaxValue}

Because Stirling’s formula is used to calculate the factorial of very large numbers, the factorial approaches \texttt{Long.MaxValue} quickly. If the factorial exceeds \texttt{Long.MaxValue}, it causes an arithmetic overflow in the computer, in which case the program continues with a strange-looking integer result, perhaps negative. Before you write the program, then, you first must write a small program that lets you determine, by trial and error, the largest value of \( n \) for which you can compute a factorial using Stirling’s formula. After you’ve determined this value, you can write the program using nested \texttt{If}s that print different messages depending on the value of \( n \). If \( n \) is within the acceptable range for your computer system, output the
number and the result with an appropriate message. If \( n \) is 0, write the message, “The number is 0. The factorial is 1.” If the number is less than 0, write: “The number is less than 0. The factorial is undefined.” If the number is greater than the largest value of \( n \) for which your computer system can compute a factorial, write: “The number is too large.”

*Suggestion:* Don’t compute Stirling’s formula directly. The values of \( n^n \) and \( e^n \) can be huge, even in floating-point form. Take the natural logarithm of the formula and manipulate it algebraically to work with more reasonable floating-point values. If \( r \) is the result of these intermediate calculations, the final result is \( e^r \). Make use of the `Math` class methods `Log` and `Exp`. These methods compute the natural logarithm and natural exponentiation, respectively.

**Case Study Follow-Up Exercises**

1. Using the source code from the `Calculator` program, identify the statements that go with Click events for the Addition button, the Subtraction button, and the Clear button.
2. Complete the original `Calculator` program by adding buttons for multiplication and division.
3. Write a test plan to test the final project and implement the test plan.