Objectives

In this tutorial, you will learn to:

■ Understand basic problem-solving techniques.
■ Understand control statements.
■ Understand and create pseudocode.
■ Use the if and if...else selection statements to choose among alternative actions.
■ Use the assignment operators.
■ Use the debugger’s Watch window.

Outline

7.1 Test-Driving the Wage Calculator Application
7.2 Algorithms
7.3 Pseudocode
7.4 Control Statements
7.5 if Selection Statement
7.6 if...else Selection Statement
7.7 Constructing the Wage Calculator Application
7.8 Assignment Operators
7.9 Formatting Text
7.10 Using the Debugger: The Watch Window
7.11 Wrap-Up

Before writing an application, it is essential to have a thorough understanding of the problem you need to solve. This will allow you to carefully plan an approach to solving the problem. When writing an application, it is equally important to recognize the types of building blocks that are available and to use proven application-construction principles. In this tutorial, you will learn the theory and principles of structured programming. Structured programming is a technique for organizing program control to help you develop applications that are easier to debug and modify. The techniques presented are applicable to most high-level languages, including C#.

7.1 Test-Driving the Wage Calculator Application

In this section, we preview this tutorial’s Wage Calculator application. This application must meet the following requirements:

Application Requirements

A payroll company calculates the gross earnings per week of employees. Employees’ weekly salaries are based on the number of hours they work and their hourly wages. Create an application that accepts this information and calculates the employee’s total (gross) earnings. The application assumes a standard work week of 40 hours. The wages for 40 or fewer hours are calculated by multiplying the employee’s hourly salary by the number of hours worked. Any time worked over 40 hours in a week is considered “overtime” and earns time and a half. Salary for time and a half is calculated by multiplying the employee’s hourly wage by 1.5 and multiplying the result of that calculation by the number of overtime hours worked. The total overtime earned is added to the user’s gross earnings for the regular 40 hours of work to calculate the total earnings for that week.

This application calculates wages from hourly salary and hours worked per week. If an employee has worked 40 or fewer hours, the employee is paid regular wages. The calculation differs if the employee has worked more than the standard 40-hour work week. In this tutorial, we introduce a programming tool
known as a **control statement** that allows us to make this distinction and perform different calculations based on different user inputs. You begin by test-driving the completed application. Then, you will learn the additional C# technologies you will need to create your own version of this application.

### Test-Driving the Wage Calculator Application

1. **Opening the completed application.** Open the C:\Examples\Tutorial07\CompletedApplication\WageCalculator directory to locate the Wage Calculator application. Double click WageCalculator.sln to open the application in Visual Studio .NET.

2. **Running the Wage Calculator application.** Select Debug > Start to run the application (Fig. 7.1). Notice that we have placed the TextBoxes vertically, rather than horizontally, in this application. To make our GUI well-organized, we have aligned the right sides of each TextBox and made the TextBoxes the same size. We have also left aligned the TextBoxes’ descriptive Labels.

![Figure 7.1 Wage Calculator application.](image)

3. **Enter the employee’s hourly wage.** Enter 10 in the Hourly wage: TextBox.

4. **Enter the number of hours the employee worked.** Enter 45 in the Weekly hours: TextBox.

5. **Calculate the employee’s gross earnings.** Click the Calculate Button. The result ($475.00) is displayed in the Gross earnings: TextBox (Fig. 7.2). Notice that the employee’s salary is the sum of the wages for the standard 40-hour work week (40 * 10) and the overtime pay (5 * 10 * 1.5).

![Figure 7.2 Calculating wages by clicking the Calculate Button.](image)

6. **Closing the application.** Close your running application by clicking its close box.

7. **Closing the IDE.** Close Visual Studio .NET by clicking its close box.

### 7.2 Algorithms

Computing problems can be solved by executing a series of actions in a specific order. A procedure for solving a problem, in terms of:

1. the actions to be executed and
2. the order in which these actions are to be executed
is called an **algorithm**. The following example demonstrates the importance of correctly specifying the order in which the actions are to be executed. Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: (1) get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast and (6) carpool to work. This routine prepares the executive for a productive day at the office.

However, suppose that the same steps are performed in a slightly different order: (1) get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our junior executive shows up for work soaking wet.

Indicating the appropriate sequence in which to execute actions is equally crucial in computer programs. **Program control** refers to the task of ordering an application’s statements correctly. In this tutorial, you will begin to investigate the program-control capabilities of C#.

### SELF-REVIEW

1. ________ refers to the task of ordering an application’s statements correctly.
   a) Actions  b) Program control  c) Control statements  d) Visual programming

2. A(n) ________ is a plan for solving a problem in terms of the actions to be executed and the order in which these actions are to be executed.
   a) chart  b) control statement  c) algorithm  d) ordered list

**Answers:** 1) b. 2) c.

### 7.3 Pseudocode

**Pseudocode** is an informal language that helps programmers develop algorithms. The pseudocode we present is particularly useful in the development of algorithms that will be converted to structured portions of C# applications. Pseudocode resembles everyday English; it is convenient and user-friendly, but it is not an actual programming language.

Pseudocode statements are not executed on computers. Rather, pseudocode helps you “think out” an application before attempting to write it in a programming language, such as C#. In this tutorial, we provide several examples of pseudocode.

The style of pseudocode that we present consists solely of characters, so you can create and modify pseudocode by using editor programs, such as the Visual Studio .NET code editor or Notepad. A carefully prepared pseudocode program can be converted easily to a corresponding C# application. Much of this conversion is as simple as replacing pseudocode statements with their C# equivalents. Let us look at an example of a pseudocode statement:

**Assign 0 to the counter**

This pseudocode statement provides an easy-to-understand task. You can put several such statements together to form an algorithm that can be used to meet application requirements. When the pseudocode algorithm has been completed, the programmer can then convert pseudocode statements to their equivalent C# statements. The pseudocode statement above, for instance, can be converted to the following C# statement:

```csharp
intCounter = 0;
```

Pseudocode normally describes only **executable statements**, which are the actions that are performed when the corresponding C# application is run. One type of programming statement that is not executed is a declaration. The declaration
int intNumber;

informs the compiler of intNumber's type and instructs the compiler to reserve space in memory for this variable. The declaration does not cause any action, such as input, output or a calculation, to occur when the application executes, so we would not include this information in the pseudocode.

SELF-REVIEW
1. Pseudocode is an artificial and informal language that helps programmers develop algorithms.
   a) Pseudocode   b) C#-Speak
   c) Notation   d) None of the above.

2. Pseudocode ________.
   a) usually describes only declarations
   b) is executed on computers
   c) usually describes only executable lines of code
   d) usually describes declarations and executable lines of code

Answers: 1) a. 2) c.

7.4 Control Statements

Normally, statements in an application are executed one after another in the order in which they are written. This is called sequential execution. However, C# allows you to specify that the next statement to be executed might not be the next one in sequence. A transfer of control occurs when an executed statement does not directly follow the previously executed statement in the written application. This is common in computer applications.

All applications can be written in terms of only three forms of control: sequence, selection and repetition. Unless directed to act otherwise, the computer executes C# statements sequentially—that is, one after the other in the order in which they appear in the application. The activity diagram in Fig. 7.3 illustrates two statements that execute in sequence. In this case, two calculations are performed in order. The activity diagram presents a graphical representation of the algorithm.

Figure 7.3 Sequence statement activity diagram.

Activity diagrams are part of the Unified Modeling Language (UML™)—an industry standard for modeling software systems. An activity diagram models the activity (also called the workflow) of a portion of a software system. Such activities may include a portion of an algorithm, such as the sequence of two statements in Fig. 7.3. Activity diagrams are composed of special-purpose symbols, such as action-state symbols (a rectangle with its left and right sides replaced with arcs curving outward), diamonds and solid circles. These symbols are connected by transition arrows, which represent the flow of the activity. Figure 7.3 does not include any diamond symbols—these will be used in later activity diagrams, beginning with Fig. 7.6.
Like pseudocode, activity diagrams help programmers develop and represent algorithms, although many programmers prefer pseudocode. Activity diagrams clearly show how control statements operate.

Consider the activity diagram for the sequence statement in Fig. 7.3. The activity diagram contains two action states, which represent actions to perform. Each action state contains an action expression—for example, “add grade to total” or “add 1 to counter”—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows in the activity diagram, called transition arrows, represent transitions, which indicate the order in which the actions represented by the action states occur. The application that implements the activities illustrated by the activity diagram in Fig. 7.3 first adds intGrade to intTotal, then adds 1 to intCounter.

The solid circle located at the top of the activity diagram represents the activity’s initial state—the beginning of the workflow before the application performs the modeled activities. The solid circle surrounded by a hollow circle that appears at the bottom of the activity diagram represents the final state—the end of the workflow after the program performs its activities.

Notice, in Fig. 7.3, the rectangles with the upper-right corners folded over, looking like sheets of paper. These are called notes in the UML. Notes are like comments in C# applications—they are explanatory remarks that describe the purpose of symbols in the diagram. Figure 7.3 uses UML notes to show the C# code that the programmer might associate with each action state in the activity diagram. A dotted line connects each note with the element that the note describes. Activity diagrams normally do not show the C# code that implements the activity, but we use notes here to show you how the diagram relates to C# code.

C# provides three types of selection statements, which we discuss in this tutorial and in Tutorial 11. The if selection statement performs (selects) an action (or sequence of actions) based on a condition. A condition is an expression with a true or false value that is used to make a decision. Conditions are evaluated (that is, tested) to determine whether their value is true or false. These values are of the bool type and are specified in C# code by using the true and false keywords. Sometimes we refer to a condition as a boolean expression, or bool expression.

If the condition evaluates to true, the actions specified by the if statement will execute. If the condition evaluates to false, the actions specified by the if statement will be skipped. The if...else selection statement performs an action (or sequence of actions) if a condition is true and performs a different action (or sequence of actions) if the condition is false. The switch statement, discussed in Tutorial 11, performs one of many actions (or sequences of actions), depending on the value of an expression.

The if statement is called a single-selection statement because it selects or ignores a single action (or a sequence of actions). The if...else statement is called a double-selection statement because it selects between two different actions (or sequences of actions). The switch statement is called a multiple-selection statement because it selects among many different actions or sequences of actions.

C# provides four types of repetition statements—while, do...while, for and foreach. The while repetition statement is covered in Tutorial 10, do...while is covered in Tutorial 11, for is covered in Tutorial 11, and foreach is covered in Tutorial 20. The words if, else, switch, while, do, for and foreach are all C# keywords. (Appendix F includes a complete list of C# keywords.)

So, C# has three forms of control—sequence, selection and repetition. Each C# application is formed by combining as many of each type of control statement as is necessary. As with the sequence of statements in Fig. 7.3, each control statement is drawn with two small circle symbols—a solid black one to represent the entry point to the control statement and a solid black one surrounded by a hollow circle to represent the exit point.
All C# control statements are **single-entry/single-exit control statement**—each has exactly one entry point and one exit point. Such control statements make it easy to build applications—the control statements are attached to one another by connecting the exit point of one control statements to the entry point of the next. This is similar to stacking building blocks, so we call it **control-statement stacking**. The only other way to connect control statements is through **control-statement nesting**, whereby one control statement can be placed inside another. Thus, algorithms in C# applications are constructed from only eight different types of control statements combined in only two ways. This is a model of simplicity.

### SELF-REVIEW

1. All C# applications can be written in terms of _______ forms of control.
   - a) one
   - b) two
   - c) three
   - d) four

2. The process of application statements executing one after another in the order in which they are written is called _______.
   - a) transfer of control
   - b) sequential execution
   - c) workflow
   - d) None of the above.

**Answers:** 1) c. 2) b.

### 7.5 if Selection Statement

A selection statement chooses among alternative courses of action in an application. For example, suppose that the passing grade on a test is 60 (out of 100). The pseudocode statement

```
If student's grade is greater than or equal to 60 
  Display "Passed"
```

determines whether the condition “student’s grade is greater than or equal to 60” is true or false. If the condition is true, then “Passed” is displayed, and the next pseudocode statement in order is “performed.” (Remember that pseudocode is not a real programming language.) If the condition is false, the display statement is ignored, and the next pseudocode statement in order is performed.

The preceding pseudocode If statement may be written in C# as

```
if ( intStudentGrade >= 60 )
  lblGradeDisplay.Text = "Passed";
```

Notice that the C# code corresponds closely to the pseudocode, demonstrating the usefulness of pseudocode as a program-development tool. The body of the if statement displays the string "Passed" in a Label. The if selection statement normally expects only one statement in its body. To include several statements in the body of an if, enclose these statements in braces { and }. A set of statements contained in a pair of braces is called a **block**. For instance, the above statement could be rewritten as

```
if ( intStudentGrade >= 60 )
{
  lblGradeDisplay.Text = "Passed";
}
```

Notice the indentation in the if statement. Such indentation enhances application readability. The C# compiler ignores whitespace characters, such as spaces, tabs and newlines, used for indentation and vertical spacing, unless the whitespace characters are contained in strings.

The if keyword must be followed by a set of parentheses that encloses a condition. The condition between the parentheses determines whether the statement(s) within the if statement will execute. If the condition is true, the body of the if
statement executes. If the condition is false, the body does not execute. Conditions in if statements can be formed by using the equality operators and relational operators (also called comparison operators), which are summarized in Fig. 7.4. The relational and equality operators all have the same level of precedence.

<table>
<thead>
<tr>
<th>Algebraic equality or relational operators</th>
<th>C# equality or relational operator</th>
<th>Example of C# condition</th>
<th>Meaning of C# condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>intX &gt; intY</td>
<td>intX is greater than intY</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>intX &lt; intY</td>
<td>intX is less than intY</td>
</tr>
<tr>
<td>≥</td>
<td>≥</td>
<td>intX &gt;= intY</td>
<td>intX is greater than or equal to intY</td>
</tr>
<tr>
<td>≤</td>
<td>≤</td>
<td>intX &lt;= intY</td>
<td>intX is less than or equal to intY</td>
</tr>
<tr>
<td>=</td>
<td>==</td>
<td>intX == intY</td>
<td>intX is equal to intY</td>
</tr>
<tr>
<td>≠</td>
<td>!=</td>
<td>intX != intY</td>
<td>intX is not equal to intY</td>
</tr>
</tbody>
</table>

**Figure 7.4** Equality and relational operators.

Figure 7.5 shows the syntax of the if statement. A statement’s syntax specifies how the statement must be formed to execute without syntax errors. Let’s look closely at the syntax of an if statement. The first line of Fig. 7.5 specifies that the statement must begin with the if keyword and be followed by a left parenthesis, (, a condition and a right parenthesis, ). Notice that we have italicized condition. This indicates that, when creating your own if statement, you should replace the text condition with the actual condition that you would like to evaluate. The second line indicates that you should replace statements with the actual statements that you want to include in the body of the if statement. These statements make up the body of the if statement. Notice that the text statements is placed within square brackets. These brackets do not appear in the actual if statement. Instead, the square brackets indicate that certain portions of the statement are optional. In this example, the square brackets indicate that all statements in the if statement’s body are optional. Of course, if there are no statements in the body of the if statement, no actions will occur as part of that statement, regardless of the condition’s value. The curly braces are required if multiple statements appear in the body of the if statement. For simplicity, we always use braces in our if statements.

**Syntax**

```csharp
if ( condition )
{
    [ statements ]
}
```

**Figure 7.5** if statement syntax.

Figure 7.6 illustrates the single-selection if statement. This activity diagram contains what is perhaps the most important symbol in an activity diagram—the diamond, or decision symbol, which indicates that a decision is to be made. Note the two sets of square brackets above or next to the arrows leading from the decision symbol—these are called guard conditions. A decision symbol indicates that the workflow will continue along a path determined by the symbol’s associated guard conditions, which can be true or false. Each transition arrow emerging from a
decision symbol has a guard condition (specified in square brackets above or next to the transition arrow). If a given guard condition is true, the workflow enters the action state to which that transition arrow points. For example, in Fig. 7.6, if the grade is greater than or equal to 60, the application displays “Passed”, then transitions to the final state of this activity. If the grade is less than 60, the application immediately transitions to the final state without displaying a message. Only one guard condition associated with a particular decision symbol can be true at once.

Figure 7.6  if single-selection statement activity diagram.

Note that the if statement (Fig. 7.6), is a single-entry/single-exit statement. The activity diagrams for the remaining control statements also contain (aside from small circle symbols and flowlines called transitions) only action-state symbols, indicating actions to be performed, and diamond symbols, indicating decisions to be made. Representing control statements in this way emphasizes the action/decision model of programming. To understand the process of structured programming better, we can envision eight bins, each containing a different type of the eight possible control statements. The control statements in each bin are empty, meaning that nothing is written in the action-state symbols and no guard conditions are written next to the decision symbols. The programmer’s task is to assemble an application, using as many control statements as the algorithm demands, combining those control statements in two possible ways (stacking or nesting) and filling in the actions and decisions (with the decisions’ guard conditions) in a manner appropriate to the algorithm.

SELF-REVIEW

1. Which of the following if statements correctly displays that a student received an A on an exam if the score was 90 or above?
   a) if ( intStudentGrade != 90 )
      {
       lblDisplay.Text = "Student received an A";
      }
   b) if ( intStudentGrade > 90 )
      {
        lblDisplay.Text = "Student received an A";
      }
   c) if ( intStudentGrade == 90 )
      {
        lblDisplay.Text = "Student received an A";
      }
   d) if ( intStudentGrade >= 90 )
      {
        lblDisplay.Text = "Student received an A";
      }

2. The _________ symbol is not a C# relational operator.
   a) <=          b) >=
   c) <>          d) >

Answers: 1) d. 2) c.
7.6 if...else Selection Statement

The if selection statement performs an indicated action (or sequence of actions) only when the condition evaluates to true; otherwise, the action (or sequence of actions) is skipped. The if...else selection statement allows the programmer to specify a different action (or sequence of actions) to be performed when the condition is true than when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
   Display "Passed"
else
   Display "Failed"
```

displays “Passed” if the student’s grade is greater than or equal to 60, but displays “Failed” if the student’s grade is less than 60. In either case, after output occurs, the next pseudocode statement in sequence is “performed.”

The preceding pseudocode If...Else statement may be written in C# as

```
if ( intStudentGrade >= 60 )
{
   lblDisplay.Text = "Passed";
}
else
{
   lblDisplay.Text = "Failed";
}
```

Note that the body of the else block is indented so that it lines up with the indented body of the if block. A standard indentation convention should be applied consistently throughout your applications. It is difficult to read programs that do not use uniform spacing conventions. The if...else selection statement follows the same general syntax as the if statement. The else keyword and any related statements are placed after the if statement as in Fig. 7.7.

**Syntax**

```
if ( condition )
{
   [ statements ]
}
else
{
   [ statements ]
}
```

**Figure 7.7** if...else statement syntax.

Figure 7.8 illustrates the flow of control in the if...else double-selection statement. Once again, note that (besides the initial state, transition arrows and final state) the only other symbols in the activity diagram represent action states and decisions. In this example, the grade is either less than 60 or greater than or equal to 60. If the grade is less than 60, the application displays "Failed". If the grade is equal to or greater than 60, the application displays "Passed". We continue to emphasize this action/decision model of computing. Imagine again a deep bin containing as many empty double-selection statements as might be needed to build any C# application. Your job as a programmer is to assemble these selection statements (by stacking and nesting) with any other control statements required by the algorithm. You fill in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm.
Nested if...else statements test for multiple conditions by placing if...else statements inside other if...else statements. For example, the following pseudocode will display “A” for exam grades greater than or equal to 90, “B” for grades in the range 80–89, “C” for grades in the range 70–79, “D” for grades in the range 60–69 and “F” for all other grades:

If student’s grade is greater than or equal to 90
Display “A”
else
  If student’s grade is greater than or equal to 80
  Display “B”
  else
    If student’s grade is greater than or equal to 70
    Display “C”
    else
      If student’s grade is greater than or equal to 60
      Display “D”
      else
        Display “F”

The preceding pseudocode may be written in C# as follows:

```csharp
if (intStudentGrade >= 90)
{
    lblDisplay.Text = "A";
} else 
{
    if (intStudentGrade >= 80)
    {
        lblDisplay.Text = "B";
    } else 
    {
        if (intStudentGrade >= 70)
        {
            lblDisplay.Text = "C";
        } else 
        {
            if (intStudentGrade >= 60)
            {
                lblDisplay.Text = "D";
            } else
            {
                lblDisplay.Text = "F";
            }
        }
    }
}
```
If intStudentGrade is greater than or equal to 90, the first condition evaluates to true and the statement lblDisplay.Text = "A"; is executed. Notice that, with a value for intStudentGrade greater than or equal to 90, the remaining three conditions will evaluate to true. These conditions, however, are never evaluated, because they are placed within the else portion of the outer if...else statement. Because the first condition is true, all statements within the else block are skipped. Let’s now assume intStudentGrade contains the value 75. The first condition is false, so the application will execute the statements within the else block of this statement. This else block also contains an if...else statement, with the condition intStudentGrade >= 80. This condition evaluates to false, causing the statements in this if...else statement’s else block to execute. This else block contains yet another if...else statement, with the condition intStudentGrade >= 70. This condition is true, causing the statement lblDisplay.Text = "C"; to execute. The else block of this if...else statement is then skipped.

Most C# programmers prefer to write the preceding if...else statement as

```csharp
if (intStudentGrade >= 90)
{
    lblDisplay.Text = "A";
}
else if (intStudentGrade >= 80)
{
    lblDisplay.Text = "B";
}
else if (intStudentGrade >= 70)
{
    lblDisplay.Text = "C";
}
else if (intStudentGrade >= 60)
{
    lblDisplay.Text = "D";
}
else
{
    lblDisplay.Text = "F";
}
```

The two statements are equivalent, but the latter statement is popular because it avoids deep indentation of the code. Such deep indentation often leaves little room on a line, forcing lines to be split and decreasing code readability. Notice that the final portion of the if...else statement uses the else keyword to handle all remaining possibilities. The else block must always be last in an if...else statement — following an else block with another else or else if is a syntax error.

**SELF-REVIEW**

1. if...else is a _________-selection statement.
   a) single  
   b) double  
   c) triple  
   d) nested

2. Placing an if...else statement inside another if...else statement is an example of _________.
   a) nesting if...else statements  
   b) stacking if...else statements  
   c) creating sequential if...else statements  
   d) None of the above.

**Answers:** 1) b. 2) a.

### 7.7 Constructing the Wage Calculator Application

The following section teaches you how to build the Wage Calculator by using the if...else statement. The if...else statement allows you to select between calculating regular wages and including overtime pay based on the number of hours
worked. The following pseudocode describes the basic operation of the *Wage Calculator* application, which runs when the user clicks **Calculate**:

When the user clicks the Calculate Button:

- Retrieve the number of hours worked and hourly wage from the TextBoxes
- If the number of hours worked is less than or equal to 40 hours
  - Gross earnings equals hours worked times hourly wage
- else
  - Gross earnings equals 40 times hourly wage plus hours above 40 times wage times 1.5

Display gross earnings

Before developing each application, you take it for a test drive. Here you interact with the application’s GUI and begin to understand the application’s purpose. You also learn the GUI components that will be required to obtain user input and display results. Frequently, when determining the requirements of an application, you will design a prototype of the application’s GUI. As you develop each application for the remainder of this book, you will use two application development aids—pseudocode and **Action/Control/Event (ACE) tables**. Pseudocode describes the algorithm—that is, the actions to be performed and the order in which those actions should be performed. As you read the pseudocode, you will see that there are specific actions to perform, such as “Calculate gross wages,” “Retrieve the number of hours worked” and “Display gross wages.” An ACE table helps relate the events that occur on GUI controls with the actions that should be performed in response to those events.

Figure 7.9 presents the Action/Control/Event (ACE) table for the *Wage Calculator* application. Sometimes, when creating an ACE table, actions in the pseudocode can be lifted and inserted directly in the left column of the table—for instance, “Display gross wages.” In other cases, one action might be represented with a substantial amount of pseudocode—for instance, calculating an employee’s gross wages requires most of the pseudocode that describes the Wage Calculator application. It would be tedious to list all this pseudocode in the table. In such cases, you might use a shorthand representation of the action, such as “Calculate gross wages.” The left column sometimes includes actions that are not represented in the pseudocode at all. For example, the action “Label the application’s controls” is not part of the pseudocode, but is an important part of constructing this application. The middle column specifies the GUI control or class associated with the action. The right column specifies the event that initiates the action.

**Action/Control/Event (ACE) Table for the Wage Calculator Application**

<table>
<thead>
<tr>
<th>Action</th>
<th>Control</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label the application's controls</td>
<td>lblWage, lblHours, lblEarnings</td>
<td>Application is run</td>
</tr>
<tr>
<td>Retrieve the number of hours worked and hourly wage from the TextBoxes</td>
<td>btnCalculate</td>
<td>Click</td>
</tr>
</tbody>
</table>
| If the number of hours worked is less than or equal to 40 hours
  - Gross earnings equals hours worked times hourly wage | txtWage, txtHours        |                              |
| else
  - Gross earnings equals 40 times hourly wage plus hours above 40 times wage times 1.5 |                          |                              |
| Display gross earnings                      | lblEarningsResult        |                              |

*Figure 7.9*  Action/Control/Event table for the *Wage Calculator* application.
The labels in the first row display information about the application to the user. These labels help guide the user through the application. In the second row, the user clicks btnCalculate to calculate the gross wages for an employee. In the third column of this row, the text Click indicates the event that initializes the calculation. The TextBoxes in the third row will obtain input from the user, accessed through the Text property. The fourth and fifth rows show the if...else statement that determines the gross wages. The final control, lblEarningsResult, is a Label that displays the application's output.

We now apply our pseudocode and the ACE table to complete the Wage Calculator application. The following box will guide you through the process of adding a Click event to the Calculate Button and declaring the variables you’ll need to calculate the employee’s wages. If you forget to add code to this Click event, the application will not respond when the user clicks the Calculate Button.

---

### Declaring Variables in the Calculate Button’s Click Event Handler

1. **Copying the template application to your working directory.** Copy the C:\Examples\Tutorial07\TemplateApplication\WageCalculator directory to your C:\SimplyCSP directory.

2. **Opening the Wage Calculator application’s template file.** Double click WageCalculator.sln in the WageCalculator directory to open the application in Visual Studio .NET.

3. **Adding the Calculate Button’s Click event handler.** In this example, the event handler calculates the gross wages when the Calculate Button’s Click event occurs. Double click the Calculate Button. The default event handler will be generated, and you will be switched to code view. Lines 170–174 of Fig. 7.10 display the generated event handler. Be sure to add the comments and break the header as shown in Fig. 7.10 so that the line numbers in your code match those presented in this tutorial.

![Generated event handler](image)

**Figure 7.10** Calculate Button event handler.

4. **Declaring variables.** As you learned in Tutorial 5, a double holds numbers with decimal points. Because hours and wages are often fractional numbers, ints are not appropriate for this application. Add lines 173–176 of Fig. 7.11 into the body of event handler btnCalculate Click. Line 174 contains a variable declaration for double dblHours, which holds the number of hours input by the user. Notice that the variable names for doubles are prefixed with dbl.

---

**Good Programming Practice**

Prefix double variable names with dbl.
5. **Declaring a constant.** Add line 178 of Fig. 7.12 to the btnCalculate_Click event handler. Line 178 contains a `constant`, a variable whose value cannot be changed after its initial declaration. Constants are declared with the `const` keyword. In this case, we assign to the `intHOUR_LIMIT` constant the maximum number of hours worked before mandatory overtime pay (40). Notice that we prefix this constant with `int` and capitalize the rest of the constant’s name to emphasize that it is a constant. Constants are useful because they are more descriptive than explicit values, such as 40. Constants can also reduce your work as a programmer. For example, if you need to change the maximum number of hours worked before mandatory overtime pay from 40 to 35, you would only change `40` to `35` once, in the constant’s declaration. If you had not used a constant, you would need to change `40` to `35` every time it appeared in your code.

6. **Saving the project.** Select `File > Save All` to save your modified code.

---

**Figure 7.11** Declaring variables of the `double` and `decimal` types.

This application introduces the `decimal` built-in type. The `decimal` type is used to store monetary amounts because it ensures rounding accuracy in arithmetic calculations involving monetary amounts. Lines 175–176 declare `decWage`, which stores the hourly wage input by the user, and `decEarnings`, which stores the total amount of earnings for the week. Notice that `decimal` variable names are prefixed with `dec`.

**Figure 7.12** Creating a constant.
Now that you have declared variables, you can use them to receive input from the user and then use that input to compute and display the user’s wages. The following box walks you through using the *if*...*else* statement to determine the user’s wages.

### Determining the User’s Wages

1. **Obtaining inputs from the TextBoxes.** Add lines 180–182 of Fig. 7.13 to the end of the *btnCalculate* `Click` event handler. Lines 181–182 assign values to `dblHours` and `decWage` from the TextBoxes in which the user enters data. Recall that `Int32.Parse` converts a string of characters to an `int`. You can use the `Double.Parse` and `Decimal.Parse` methods to convert a string of characters to a `double` and to a `decimal`, respectively (lines 181–182).

   ![Variable assignment](image)

   **Figure 7.13** Assigning data to variables.

2. **Determining wages based on hours worked.** Begin to add the *if*...*else* statement shown in lines 184–199 of Fig. 7.14 to the end of the *btnCalculate* `Click` event handler. You might need to indent as you go. This *if*...*else* statement determines whether employees earn overtime in addition to their usual wages. Line 185 determines whether the value stored in `dblHours` is less than or equal to `intHOUR_LIMIT`. If it is, then line 188 assigns the value of the product of the hours and the wage to `decEarnings`. C# does not allow you to multiply a variable of the `double` type by a variable of the `decimal` type. The variables must have compatible types. Because the product will be assigned to `decEarnings`, a variable of type `decimal`, it makes sense to convert `dblHours` to a `decimal`. The cast operator is used to perform the conversion in line 188.

   If, on the other hand, `dblHours` is not less than or equal to `intHOUR_LIMIT`, then the application proceeds to the *else* keyword in line 189. Line 193 computes the wage for the hours worked up to the limit set by `intHOUR_LIMIT` and assigns it to `decEarnings`. No cast operator is required because an `int` can be implicitly converted to a `decimal`. Line 197 determines how many hours over `intHOUR_LIMIT` there are (by using the expression `dblHours - intHOUR_LIMIT`) and then converts the result to a `decimal`. Line 198 calculates the user’s time-and-a-half pay for overtime hours, which is 1.5 times the user’s hourly wages. The suffix `M` makes the value 1.5 a `decimal`. With no such suffix, the value 1.5 is treated as a `double`. This suffix is another way to convert a value’s type. This method can be applied to actual values, such as 1.5, but not to variables. The cast operator can be applied to either values or variables. The `decimal` result in line 197 is multiplied by the `decimal` result in line 198. This product is then added to the value of `decEarnings`, and the result is assigned to `decEarnings` (line 196).
3. **Displaying the result.** Add lines 201–202 of Fig. 7.15 to the end of the btnCalculate_Click event handler. Line 202 assigns the value in decEarnings to the Text property of the lblEarningsResult Label, using the Convert.ToString method to convert decEarnings from a decimal to a string.

4. **Running the application.** Select Debug > Start to run your application (Fig. 7.16). Enter 10 in the Hourly wage: TextBox, and enter 45 in the Weekly Hours: TextBox. Press the Calculate Button. Notice that the output is not yet formatted as it should be in the completed application. We will demonstrate how to add this functionality later in this tutorial.

5. **Closing the application.** Close your running application by clicking its close box.
SELF-REVIEW

1. The decimal type is used to store ________.
   a) letters and digits  
   b) integers  
   c) strings  
   d) monetary amounts

2. Constants are declared with the ________ keyword.
   a) fixed  
   b) constant  
   c) final  
   d) const

**Answers:** 1) d. 2) d.

### 7.8 Assignment Operators

C# provides several assignment operators for abbreviating assignment statements. For example, the statement

```csharp
intValue = intValue + 3;
```
which adds 3 to the value in `intValue`, can be abbreviated with the addition assignment operator, `+=`, as

```csharp
intValue += 3;
```

The `+=` operator adds the value of the right operand to the value of the left operand and stores the result in the left operand. C# provides assignment operators for several binary operators, including `+`, `-`, `*` and `/`. When an assignment statement is evaluated, the expression to the right of the operator is always evaluated first, then assigned to the variable on the left. C# also provides special increment (`++`) and decrement (`--`) operators. The `++` operator increases its left operand by one. The `--` operator decreases its left operand by one. Both are unary operators. When the `++` and `--` operators are written immediately to the right of the operand (as in `intC++`), they are called the **postfix increment and decrement operators**, respectively. When these operators are written immediately to the left of the operand (as in `++intC`), they are called the **prefix increment and prefix decrement operators**. If a prefix operator is used, its operand will be incremented or decremented by 1, after which the new value of the operand is used in the expression in which it appears. If a postfix operator is used, its operand will be used in the expression in which it appears, after which the operand’s value is incremented or decremented by 1. We predominantly use the postfix operators in this text. Figure 7.17 includes the assignment operators, sample expressions using these operators and explanations.
The following box demonstrates abbreviating our time-and-a-half calculation with the `+=` operator. When you run the application again, you will notice that the program runs the same as before—all that has changed is that one of the longer statements was made shorter.

**Using the Addition Assignment Operator**

1. **Adding the addition assignment operator.** Replace lines 196–198 of Fig. 7.14 with lines 196–197 of Fig. 7.18. In this step, we have used the addition assignment operator to make our statement shorter. Notice that the statement still performs the same action—the time-and-a-half pay for the user is calculated and added to the regular wages earned.

2. **Running the application.** Select **Debug > Start** to run your application. Notice that the application executes as it did in the last box.

3. **Closing the application.** Close your running application by clicking its close box.

**SELF-REVIEW**

1. The `*=` operator ________.
   a) squares the value of the right operand and stores the result in the left operand
   b) adds the value of the right operand to the value of the left operand and stores the result in the left operand
   c) creates a new variable and assigns the value of the right operand to that variable
   d) multiplies the value of the left operand by the value of the right operand and stores the result in the left operand
2. If intX is initialized with the value 5, what value will intX contain after the expression intX = 3; is executed?
   a) 3  
   b) 5  
   c) 7  
   d) 2

   **Answers:** 1) d. 2) d.

### 7.9 Formatting Text

There are several ways to format output in C#. In this section, we introduce the `String.Format` method to control how text displays. Modifying the appearance of text for display purposes is known as **text formatting**. This method takes as an argument a **format control string**, followed by arguments that indicate the values to be formatted. The format control string argument specifies how the remaining arguments are to be formatted.

Recall that your **Wage Calculator** does not display the result of its calculation with the appropriate decimal and dollar sign that you saw when test-driving the application. In the following box, you learn how to apply currency formatting to the value in the **Gross earnings** TextBox.

#### Formatting the Gross Earnings

1. **Modifying the Calculate Button’s Click event.** Replace line 202 of Fig. 7.15 with lines 201–202 of Fig. 7.19.

   ```csharp
   Line 201 sends the format control string, “{0:C}”, and the value to be formatted, decEarnings, to the String.Format method. The 0 indicates that argument zero (decEarnings—the first argument after the format control string) should take the format specified by the letter after the colon; this letter is called the **format specifier**. In this case, we use the format defined by the uppercase letter C, which represents the **currency format**, used to display values as monetary amounts. The effect of the C format specifier varies, depending on the locale setting of your computer. In our case, the result is preceded with a dollar sign ($) and displayed with two decimal places (representing cents), because we are in the United States.

   2. **Running the application.** Select Debug > Start to run your application. Notice that the output in the **Gross Earnings** TextBox is properly formatted as in the completed application.

   3. **Closing the application.** Close your running application by clicking its close box.
Figure 7.20 shows several format specifiers. All format specifiers are case insensitive, so the uppercase letters may be used interchangeably with their lowercase equivalents. Note that format code D may be used only with ints.

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Currency. Formats the currency based on the computer’s locale setting. For U.S. currency, precedes the number with $, separates every three digits (to the left of the decimal place) with commas and sets the number of decimal places to two.</td>
</tr>
<tr>
<td>E</td>
<td>Scientific notation. Displays one digit to the left of the decimal point and six digits to the right of the decimal point, followed by the character E and a three-digit integer representing the exponent of a power of 10. For example, 956.2 is formatted as 9.562000E+002.</td>
</tr>
<tr>
<td>F</td>
<td>Fixed point. Sets the number of decimal places to two by default.</td>
</tr>
<tr>
<td>G</td>
<td>General. C# chooses either E or F for you, depending on which representation generates a shorter string.</td>
</tr>
<tr>
<td>D</td>
<td>Decimal integer. Displays an integer as a whole number in standard base-10 format.</td>
</tr>
<tr>
<td>N</td>
<td>Number. Separates every three digits with a comma and sets the number of decimal places to two by default.</td>
</tr>
</tbody>
</table>

Figure 7.21 presents the source code for the Wage Calculator application. The lines of code that contain new programming concepts that you learned in this tutorial are highlighted.

```csharp
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WageCalculator
{
    /// <summary>
    /// Summary description for FrmWageCalculator.
    /// </summary>
    public class FrmWageCalculator : System.Windows.Forms.Form
    {
        // Label and TextBox for hourly wage
        private System.Windows.Forms.Label lblWage;
        private System.Windows.Forms.TextBox txtWage;

        // Label and TextBox for weekly hours
        private System.Windows.Forms.Label lblHours;
        private System.Windows.Forms.TextBox txtHours;

        // Labels to display gross earnings
        private System.Windows.Forms.Label lblEarnings;
        private System.Windows.Forms.Label lblEarningsResult;

        // Button to calculate total earnings
    }
}
```

Figure 7.21 Wage Calculator application code. (Part 1 of 3.)
```csharp
/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

class FrmWageCalculator
{
    // Required for Windows Form Designer support
    InitializeComponent();

    // TODO: Add any constructor code after InitializeComponent call

    // Windows Form Designer generated code
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.Run( new FrmWageCalculator() );
    }

    // handles Click event
    private void btnCalculate_Click(object sender, System.EventArgs e)
    {
        // declare variables
        double dblHours;
        decimal decWage;
        decimal decEarnings;

        // assign values from user input
        dblHours = Double.Parse( txtHours.Text );
        decWage = Decimal.Parse( txtWage.Text );

        const int intHOUR_LIMIT = 40; // declare constant
    }

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    protected override void Dispose( bool disposing )
    {
        if( disposing )
        {
            if (components != null)
            {
                components.Dispose();
            }
            base.Dispose( disposing );
        }
    }
```
SELF-REVIEW

1. The `String.Format` method is used to
   a) create constant variables       b) control how text is formatted
   c) format C# statements           d) All of the above.

2. The ______ format displays values as monetary amounts.
   a) monetary      b) cash
   c) currency      d) dollar

**Answers:** 1) b. 2) c.

7.10 Using the Debugger: The Watch Window

Visual Studio .NET includes several debugging windows that are accessible from the **Debug > Windows** submenu. The **Watch** window, which is available only in break mode, allows the programmer to examine the value of a variable or expression. You can use the **Watch** window to view changes in a variable’s value as the application executes, or you can change a variable’s value yourself by entering the new value directly into the **Watch** window. Each expression or variable that is added to the **Watch** window is called a **watch**. In the following box, we demonstrate how to add, remove and manipulate watches by using the **Watch** window.
1. **Starting debugging.** If the IDE is not in code view, switch to code view now. Set breakpoints at lines 182 and 188 (Fig. 7.22). Select **Debug > Start** to run the application. The **Wage Calculator** Form appears. Enter 12 into the **Hourly wage:** TextBox and 40 into the **Weekly hours:** TextBox. Click the **Calculate** Button.

![Figure 7.22 Breakpoints added to Wage Calculator application.](image)

2. **Suspending application execution.** Clicking the **Calculate** Button will cause **btnCalculate_Click** event handler to run until the breakpoint is reached. When the breakpoint is reached, application execution is paused, and the IDE switches into break mode. Notice that the active window has been changed from the running application to the IDE (Fig. 7.23). The **Wage Calculator** application is still running, but it is hidden behind the IDE.

![Figure 7.23 Suspending application execution.](image)
3. **Examining data.** Once the application has entered break mode, you are free to explore the values of various variables, using the debugger’s Watch window. To display the Watch window, select Debug > Windows > Watch > Watch 1. Notice that there are actually four options in the Debug > Windows > Watch menu—Watch 1, Watch 2, Watch 3 and Watch 4. Each window provides the same functionality. The four options simply allow you to have several Watch windows open at once. This enables you to display data side-by-side or to set the different Watch windows to display data in different formats. The Watch window (Fig. 7.24) is initially empty. To add a watch, you can type an expression into the Name column. Single click in the first field of the Name column. Type dblHours, then press Enter. The value and type will be added by the IDE (Fig. 7.24). Notice that this value is 40.0—the value assigned to dblHours in line 182. Type decWage in the next row, then press Enter. The value displayed for decWage is 0. The type of decWage is listed as System.Decimal because the decimal type is defined in the System namespace.

![Figure 7.24 Watch window.](image)

4. **Examining different expressions.** Add the expression `(dblHours + 3) * 5` into the Watch 1 window, then press Enter. Notice that the Watch window can evaluate arithmetic expressions, returning the value 215.0. Add the expression `decWage == 3` into the Watch 1 window. Expressions containing the `==` symbol are treated as bool expressions. The value returned is false, because decWage does not currently contain the value 3. Add the expression `intValueThatDoesNotExist` into the Watch 1 window. This identifier does not exist in the current application and therefore cannot be evaluated. An appropriate message is displayed in the Value field. Your Watch window should look similar to Fig. 7.25.

![Figure 7.25 Examining expressions.](image)

5. **Removing an expression from the Watch window.** At this point, we would like to clear the final expressions from the Watch window. To remove an expression, simply right click the expression in the Watch window and select Delete Watch (Fig. 7.26). Remove the three expressions that you added in Step 4.
6. **Viewing modified values in a Watch window.** Continue debugging by selecting **Debug > Continue.** The application will continue to execute until the next breakpoint, at line 188. Line 182 executes, assigning the wage value entered (12) to `decWage`. The `if` continuation condition evaluates to `true` in line 185, and the application is once again suspended in line 188. Notice that the value of `decWage` has changed not only in the application, but in the **Watch 1** window as well. Because the value has changed since the last time the application was suspended, the modified value is displayed in red (Fig. 7.27).

7. **Modifying values directly in a Watch window.** The **Watch** window can be used to change the value of a variable simply by entering the new value in the **Value** column. Click in the **Value** field for `dblHours`, replace 40.0 with 10.0, then press **Enter.** The modified value appears in red (Fig. 7.28). This option enables you to test various values to confirm the behavior of your application. If you repeat this exercise multiple times in the same application, a dialog (Fig. 7.29) might appear. Simply disregard this dialog by clicking the **Continue Button**.
8. **Viewing the application result.** Select Debug > Continue to continue pro-
gram execution. The btnCalculate_Click event handler finishes execution and displays the result in a Label. Notice that the result is $120.00, because we changed dblHours to 10.0 in the last step. The TextBox to the right of Weekly hours: still displays the value 40, because we changed the value of dblHours, but not the Text property of either TextBox. Once the application has finished running, the focus is returned to the Wage Calculator window, and the final results are displayed (Fig. 7.30).

![Wage Calculator](image)

**Figure 7.30** Output displayed after the debugging process.

9. **Closing the application.** Close your running application by clicking its close box.

10. **Closing the IDE.** Close Visual Studio .NET by clicking its close box.

---

**SELF-REVIEW**

1. An application enters break mode when _________.
   a) Debug > Start is selected  
   b) a breakpoint is reached  
   c) the Watch window is used  
   d) there is a syntax error

2. The Watch window allows you to _________.
   a) change variable values  
   b) view variable type information  
   c) evaluate expressions  
   d) All of the above.

**Answers:** 1) b. 2) d.

---

**7.11 Wrap-Up**

In this tutorial, we discussed techniques for solving programming problems. We introduced algorithms, pseudocode, the UML and control statements. We discussed different forms of control and when each might be used.

You began by test-driving an application that used the if...else statement to determine an employee’s weekly pay. You studied different control statements and used the UML to diagram the decision-making processes of the if and if...else statements.

You learned how to format text by using the String.Format method and how to abbreviate mathematical statements by using the assignment operators.

In the Using the Debugger section, you learned how to use the Watch window to view an application’s data. You learned how to add watches, remove watches and change variable values.

In the next tutorial you will learn how to display message dialogs based on user input. You will study the logical operators, which give you more expressive power for forming the conditions in your control statements. You will use the CheckBox control to allow the user to select from various options in a dental payment application.
**SKILLS SUMMARY**

Choosing Among Alternate Courses of Action
- Use the `if` or `if...else` statements.

Conceptualizing the Application Before Using Visual Studio .NET
- Use pseudocode.
- Create an Action/Control/Event (ACE) table.

Understanding Control Statements
- View the control statement’s corresponding UML diagram.

Performing Comparisons
- Use the equality and relational operators.

Creating a Constant
- Use the `const` keyword.
- Assign a value to the constant in the declaration.

Abbreviating Assignment Expressions
- Use the assignment operators.

Formatting a Value as a Monetary Amount
- Use the format code `C` in method `String.Format`.

Examining Data During Application Execution
- Use the debugger to set a breakpoint, and examine the `Watch` window.

**KEY TERMS**

**ACE (Action/Control/Event) table** — A program development tool you can use to relate GUI events with the actions that should be performed in response to those events.

**action/decision model of programming** — A model representing control statements as UML activity diagrams with rounded rectangles, indicating actions to be performed, and diamond symbols, indicating decisions to be made.

**action expression (in the UML)** — Used in an action state within a UML activity diagram to specify a particular action to perform.

**action state** — An action to perform in a UML activity diagram that is represented by an action-state symbol.

**action-state symbol** — A rectangle with its left and right sides replaced with arcs curving outward that represents an action to perform in a UML activity diagram.

**activity diagram** — A UML diagram that models the activity (also called the workflow) of a portion of a software system.

**algorithm** — A procedure for solving a problem, specifying the actions to be executed and the order in which these actions are to be executed.

**assignment operators** — Operators used for abbreviating assignment statements.

**block** — A group of code statements.

**bool type** — Has the value `true` or `false`.

**comparison operators** — See relational operators.

**condition** — An expression with a true or false value that is used to make a decision.

**const keyword** — Declares a constant, which is a variable whose value cannot be changed after its initial declaration.

**constant** — A variable whose value cannot be changed after its initial declaration.

**control statement** — An application component that specifies the order in which statements execute (also known as the flow of control).

**control-statement nesting** — Placing one control statement in the body of another control statement.

**control-statement stacking** — A set of control statements in sequence. The exit point of one control statement is connected to the entry point of the next control statement in sequence.
currency format — Used to display values as monetary amounts.
decimal type — Used to store monetary amounts.
Decimal.Parse method — Converts a given string of characters to a value of type decimal.
decision symbol — The diamond-shaped symbol in a UML activity diagram that indicates that a decision is to be made.
diamond — A symbol (also known as the decision symbol) in a UML activity diagram; that indicates that a decision is to be made.
dotted line — A UML activity diagram symbol that connects each UML-style note with the element that the note describes.
double-selection statement — A statement, such as if...else, that selects between two different actions or sequences of actions.
Double.Parse method — Converts the given string of characters to a value of type double.
equality operator — Operator that compares two values. Returns true if the two values are equal; otherwise, returns false.
executable statement — Actions that are performed when the corresponding C# application is run.
false keyword — A bool value that represents a condition that is false.
format control string — A string that specifies how data should be formatted.
format specifier — Code that specifies the type of format that should be applied to a string for output.
final state — Represented by a solid circle surrounded by a hollow circle in a UML activity diagram—the end of the workflow after a program performs its activities.
guard condition — An expression contained in square brackets above or next to the arrows leading from a decision symbol in a UML activity diagram that determines whether workflow continues along a path.
if selection statement — Performs an action (or sequence of actions) based on a condition. This is also called a single-selection statement.
if...else selection statement — Performs an action (or sequence of actions) if a condition is true and performs a different action (or sequence of actions) if the condition is false. This is also called a double-selection statement.
initial state — The beginning of the workflow in a UML activity diagram before the program performs the modeled activities.
multiple-selection statement — Selects from among many different actions or sequences of actions.
nested statement — A control statement placed inside another control statement.
note — An explanatory remark (represented by a rectangle with a folded upper-right corner) describing the purpose of a symbol in a UML activity diagram.
postfix increment and decrement operators — The ++ and -- operators, when they appear to the right of the operand. Causes the operand’s value to be used in the expression in which the operand appears, after which the value of the operand is incremented or decremented by 1.
prefix increment and decrement operators — The ++ and -- operators, when they appear to the left of the operand. Causes the operand’s value to be incremented or decremented by 1, after which the operand’s value is used in the expression in which the operand appears.
program control — The task of ordering an application’s statements in the correct order.
pseudocode — An informal language that helps programmers develop algorithms.
relational operators — Operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) that compare two values.
repetition statement — Allows the programmer to specify that an action or sequence of actions should be repeated, depending on the value of a condition.
selection statement — Selects among alternative courses of action.
sequence statement — Built into C#—unless directed to act otherwise, the computer executes C# statements sequentially.
sequential execution — Statements in an application are executed one after another in the order in which they are written.
single-entry/single-exit control statement—A control statement that has one entry point and one exit point. All C# control statements are single-entry/single-exit control statements.

single-selection statement—A control statement that selects or ignores a single action or sequence of actions.

small circles (in the UML)—The solid circle in an activity diagram represents the activity’s initial state and the solid circle surrounded by a hollow circle represents the activity’s final state.

solid circle (in the UML)—Symbol that represents the activity’s initial state.

String.Format method—Formats a string.

structured programming—A technique for organizing program control that helps you develop applications that are easy to understand, debug and modify.

syntax—Specifies how a statement must be formed to execute without syntax errors.

text formatting—Modifying the appearance of text for display purposes.

transfer of control—Occurs when an executed statement does not directly follow the previously executed statement in the written application.

transition—A change from one action state to another, represented by transition arrows in a UML activity diagram.

transition arrow (in the UML)—Symbol that represents a transition.

true keyword—A bool value that represents a condition that is true.

UML (Unified Modeling Language)—An industry standard for modeling software systems graphically.

watch—An expression or variable that is added to the Watch window.

Watch window—A Visual Studio .NET window that allows you to view variable values as an application is being debugged.

workflow—The activity of a portion of a software system.

GUI DESIGN GUIDELINES

Overall Design
- Format all monetary amounts using the C (currency) format specifier.

TextBox
- When using multiple TextBoxes vertically, align the TextBoxes on their right sides, and where possible make the TextBoxes the same size. Left-align the descriptive Labels for such TextBoxes.

CONTROLS, EVENTS, PROPERTIES & METHODS

Decimal Represents a monetary value.
  - Method
    Parse—Converts the given string of characters to a value of type decimal.

Double Represents a floating-point number (one with a decimal point).
  - Method
    Parse—Converts the given string of characters to a value of type double.

String Represents a series of characters treated as a single unit.
  - Method
    Format—Arranges the string of characters in a specified format.

MULTIPLE-CHOICE QUESTIONS

7.1 The ________ operator returns false if the left operand is larger than the right operand.
   a) ==
   b) <
   c) <=
   d) All of the above.
7.2 A ______ occurs when an executed statement does not directly follow the previously executed statement in the written application.
   a) transition  b) flow  c) logical error  d) transfer of control

7.3 A variable or an expression that is added to the Watch window is known as a ______.
   a) watched variable  b) watched expression  c) watch  d) watched value

7.4 The if statement is called a ______ statement because it selects or ignores one action or sequence of actions.
   a) single-selection  b) multiple-selection  c) double-selection  d) repetition

7.5 The three types of control statements are the sequence statement, the selection statement and the ______ statement.
   a) repeat  b) looping  c) redo  d) repetition

7.6 In an activity diagram, a rectangle with curved sides represents ______.
   a) a complete algorithm  b) a comment  c) an action  d) the termination of the application

7.7 The if…else selection statement ends with a(n) ______.
   a) right brace (})  b) endif statement  c) endelse statement  d) double-selection statement

7.8 A variable of type bool can be assigned the ______ keyword or the ______ keyword.
   a) true, false  b) off, on  c) true, notTrue  d) yes, no

7.9 A variable whose value cannot be changed after its initial declaration is called a ______.
   a) double  b) constant  c) standard  d) bool

7.10 The ______ operator assigns the result of adding the left and right operands to the left operand.
    a) +  b) ++  c) +=  d) ++

**EXERCISES**

7.11 *(Currency Converter Application)* Develop an application that functions as a currency converter (Fig. 7.31). Users must provide a number in the Dollars: TextBox and a currency name (as text) in the Convert from Dollars to: TextBox. Clicking the Convert Button will convert the specified amount into the indicated currency and display it in a Label. Limit yourself to the following currencies as user input: Dollars, Euros, Yen and Pesos. Use the following exchange rates: 1 Dollar = 1.02 Euros, 120 Yen and 10 Pesos. Use the M suffix to convert 1.02 to a decimal.

![Currency Converter GUI](Image)
a) **Copying the template to your working directory.** Copy the directory C:\Examples\Tutorial07\Exercises\CurrencyConverter to your C:\SimplyCSP directory.

b) **Opening the application’s template file.** Double click CurrencyConverter.sln in the CurrencyConverter directory to open the application.

c) **Add an event handler for the Convert Button’s Click event.** Double click the Convert Button to generate an empty event handler for the Button’s Click event. The code for Steps d–f belongs in this event handler.

d) **Obtaining the user input.** Use the Decimal.Parse method to convert the user input from the Dollars: TextBox to a decimal. Assign the decimal to a variable decAmount.

e) **Performing the conversion.** Use an if...else statement to determine which currency the user entered. Assign the result of the conversion to decAmount.

f) **Displaying the result.** Display the result using the String.Format method with the F format specifier.

g) **Running the application.** Select Debug > Start to run your application. Enter a value in dollars to convert and the currency you wish to convert to. Click the Convert Button and, using the specified exchange rates, verify that the correct output is displayed.

h) **Closing the application.** Close your running application by clicking its close box.

i) **Closing the IDE.** Close Visual Studio .NET by clicking its close box.

### 7.12 (Expanded Wage Calculator that Performs Tax Calculations)

Develop an application that calculates an employee’s wages (Fig. 7.32). The user should provide the hourly wage and number of hours worked per week. When the Calculate Button is clicked, display the gross earnings in the Gross earnings: TextBox. The Less FWT: TextBox should display the amount deducted for Federal taxes and the Net earnings: TextBox should display the difference between the gross earnings and the Federal tax amount. Assume overtime wages are 1.5 times the hourly wage and Federal taxes are 15% of gross earnings. The Clear Button clears all fields.

![Expanded Wage Calculator GUI](image)

**Figure 7.32** Expanded Wage Calculator GUI.

a) **Copying the template to your working directory.** Copy the directory C:\Examples\Tutorial07\Exercises\ExpandedWageCalculator to your C:\SimplyCSP directory.

b) **Opening the application’s template file.** Double click wageCalculator.sln in the ExpandedWageCalculator directory to open the application.

c) **Modifying the Calculate Button’s Click event handler.** Add the code for Steps d–f to btnCalculate_Click.

d) **Adding a new variable.** Declare decFederalTaxes to store the amount deducted for Federal taxes.

e) **Calculating and displaying the Federal taxes deducted.** Multiply the total earnings (decEarnings) by 0.15 (that is, 15%) to determine the amount to be removed for taxes. Use the M suffix to convert 0.15 to a decimal. Assign the result to decFederalTaxes. Display this value using the String.Format method with the C format specifier.

f) **Calculating and displaying the employee’s net pay.** Subtract decFederalTaxes from decEarnings to calculate the employee’s net earnings. Display this value using String.Format method with the C format specifier.

g) **Creating an event handler for the Clear Button.** Double click the Clear Button to generate an empty event handler for the Click event. This event handler should clear user input from the two Text Boxes and the results from the three Labels.
h) **Running the application.** Select Debug > Start to run your application. Enter an hourly wage and the number of hours worked. Click Calculate and verify that the appropriate output is displayed for gross earnings, amount taken out for federal taxes and the net earnings. Click the Clear Button and check that all fields are cleared.

i) **Closing the application.** Close your running application by clicking its close box.

j) **Closing the IDE.** Close Visual Studio .NET by clicking its close box.

### 7.13 (Credit Checker Application)

Develop an application that determines whether a department-store customer has exceeded the credit limit on a charge account (Fig. 7.33). Each customer enters an account number (an int), a balance at the beginning of the month (a decimal), the total of all items charged for the month (a decimal), the total of all credits applied to the customer’s account for the month (a decimal), and the customer’s allowed credit limit (a decimal). The application should input each of these facts, calculate the new balance \((\text{beginning balance} - \text{credits} + \text{charges})\), display the new balance and determine whether the new balance exceeds the customer’s credit limit. If the customer’s credit limit is exceeded, the application should display a message (in a Label at the bottom of the Form) informing the customer of this fact.

![Credit Checker GUI](image)

**Figure 7.33** Credit Checker GUI.

a) **Copying the template to your working directory.** Copy the directory C:\Examples\Tutorial07\Exercises\CreditChecker to your C:\SimplyCSP directory.

b) **Opening the application’s template file.** Double click CreditChecker.sln in the CreditChecker directory to open the application.

c) **Adding the Calculate Button’s Click event handler.** Double click the Calculate Button to generate the empty event handler for the Click event. The code for Steps d–g is added to this event handler.

d) **Declaring variables.** Declare an int variable to store the account number. Declare four decimal variables to store the starting balance, charges, credits and credit limit. Declare a fifth decimal variable to store the new balance in the account after the credits and charges have been applied.

e) **Obtaining user input.** Obtain the user input from the TextBoxes’ Text properties.

f) **Calculating and displaying the new balance.** Calculate the new balance by subtracting the total credits from the starting balance and adding the charges. Assign the result to a variable. Display the result formatted as currency.

g) **Determining if the credit limit has been exceeded.** If the new balance exceeds the specified credit limit, a message should be displayed in lblError.

h) **Handling the Account number: TextBox's TextChanged event.** Double click the Account number: TextBox to create its TextChanged event handler. This event handler should clear the other TextBoxes, the error message Label and the result Label.

i) **Running the application.** Select Debug > Start to run your application. Enter an account number, your starting balance, the amount charged to your account, the amount credited to your account and your credit limit. Click the Calculate Balance Button and verify that the new balance displayed is correct. Enter an amount
charged that exceeds your credit limit. Click the Calculate Balance Button and ensure that a message is displayed in the lower Label.

j) **Closing the application.** Close your running application by clicking its close box.

k) **Closing the IDE.** Close Visual Studio .NET by clicking its close box.

---

**What does this code do?**

Assume that `txtAge` is a TextBox control and that the user has entered the value 27 into this TextBox. Determine the action performed by the following code:

```csharp
int intAge;
intAge = Int32.Parse( txtAge.Text );
if ( intAge < 0 )
{
    txtAge.Text = "Enter a value greater than or equal to zero."
}
else if ( intAge < 13 )
{
    txtAge.Text = "Child"
}
else if ( intAge < 20 )
{
    txtAge.Text = "Teenager"
}
else if ( intAge < 30 )
{
    txtAge.Text = "Young adult"
}
else if ( intAge < 65 )
{
    txtAge.Text = "Adult"
}
else
{
    txtAge.Text = "Senior Citizen"
}
```

---

**What's wrong with this code?**

Assume that `lblAMPM` is a Label control. Find the error(s) in the following code.

```csharp
int intHour;
intHour = 14;
if ( intHour < 0 )
{
    lblAMPM.Text = "Time Error."
}
else if ( intHour > 23 )
{
    lblAMPM.Text = "Time Error."
}
else
{
    lblAMPM.Text = "PM"
}
else if ( intHour < 12 )
{
    lblAMPM.Text = "AM"
}
```
Using the Debugger

7.16 (Grade Calculator Application) Copy the directory C:\Examples\Tutorial07\Exercises\Debugger\Grades into your C:\SimplyCSP directory. This directory contains the Grades application, which takes a number from the user and displays the corresponding letter grade. For values in the range 90–100 it should display A, for 80–89, B, for 70–79, C, for 60–69, D and for anything lower, F. Run the application. Enter the value 85 in the TextBox and click Calculate. Notice that the application displays D when it ought to display B. Select View > Code to enter the code editor and set as many breakpoints as you feel necessary. Select Debug > Start to use the debugger to help you find the error(s). Figure 7.34 shows the incorrect output when the value 85 is input.

![Figure 7.34 Incorrect output for Grade Calculator application.](image)

Programming Challenge

7.17 (Encryption Application) A company transmits data over the telephone, but it is concerned that its phones could be tapped. All its data is transmitted as four-digit ints. The company has asked you to write an application that encrypts its data so that it may be transmitted more securely. Encryption is the process of transforming data for security reasons. Create a Form similar to Fig. 7.35. Your application should read four digits entered by the user and encrypt the information as follows:

- a) Replace each digit by \( \text{(the sum of that digit plus 7) modulo 10} \). We use the term modulo to indicate you are to use the remainder (%) operator.

- b) Swap the first digit with the third, and swap the second digit with the fourth.

![Figure 7.35 Encryption application.](image)