2

Control structures, classes, and data types

Objectives

In this session you will learn

- The elements of the structure theorem and how they are used to construct algorithms;
- How to define a classes and create objects in a C# program;
- How to define attributes and operations in classes while maintaining encapsulation;
- How to produce complex formatted output using composite formatting;

Prescribed Reading: Chapter 3 finishes with an introduction of basic decision making with the if statement. Finish the chapter by reading Section 3.9.

Prescribed Reading: Chapter 4 introduces how to develop applications using several objects, by describing how to define classes, instantiate them (i.e., create objects), and how to define the basic elements of classes such as instance attributes and methods.

Prescribed Reading: Chapter 5 and Chapter 6 continue coverage of control structures in the C# programming language, which you should already be familiar with from the pre-requisite unit, thus only a cursory reading of these chapters should be required.

Prescribed Reading: Appendix G, an online resource provided with the textbook, provides an introduction to using the Visual Studio debugger, a critical skill for developing more complex software that you will use many times during the trimester and the future.

2.1. Introduction

We began our study of object-oriented development in Session 1, however we have only begun writing the simplest of applications, in particular those involving variables, console input/output, arithmetic expressions, and simple decisions using if statements. In this session, we continue our study by reviewing control structures from the pre-requisite unit and how they are used to construct algorithms. We will learn how to define classes and create objects, the fundamental building blocks for object-oriented applications. We will then extend our study of formatted output from Section 1.7 by considering composite format strings, before finishing with an examination of logic errors and debugging.

2.2. Control Structures and the Structure Theorem

In object-oriented development, we decompose (break-down) an application into its constituent parts, known as objects. However even after this decomposition there are often still a number of problems to be solved. Consider the following possible operations defined in a class:

- Calculate the total cost of an invoice;
- Sorting a list of names in an address book;
- Reconstructing a file downloaded using BitTorrent;
- Encrypting a user's password for storage in a web site's database; or
- Determine the path for a computer-driven character to move in a game.

This is just a short list of example problems you may be required to solve in a programming career. We solve such problems by defining one or more ***algorithms*** that perform the required functionality. An algorithm is a series of steps or actions that are performed in a particular sequence or order. A good analogy for an algorithm can be seen in a cooking recipe, which tells you how to cook a meal by combining various food items together, in a specific order, using one or more techniques. The recipe (algorithm) is the sequence of steps to convert the food items (input) into the meal (output).

The ***structure theorem***[1] tells us that we can express algorithms to solve any problem using a combination of three elements:

1. Sequence – there is an ordering imposed upon the steps of the algorithm;
2. Selection – making decisions on the basis of some condition, and then choose to perform/not to perform one or more of the steps of the algorithm; and
3. Repetition – performing one or more steps of the algorithm a number of times until some condition is achieved/no longer maintained.

In this section we review these concepts from the pre-requisite units, beginning with an examination of how to express conditions in C# before reviewing the main selection and repetition control structures.

2.2.1. Conditions

Both selection and repetition control structures often make use of a ***condition***. A condition is an expression that will evaluate to a boolean result, i.e., a value of either `true` or `false`. The expression is used to compare one value against another and involve either an ***equality operator*** or a ***relational operator***. The values used in the comparison could be stored in a variable, specified by a literal or constant, or could be a result of some operation, e.g., a return value from a method call. Equality operators are used to compare two values to determine whether they are equal (==) or not equal (`!=`). For example, to test the age of a person (stored in a variable `age`) being equal or not equal to `18` (a literal) would result in the following conditions:

```
age == 18              // age is equal to 18
age != 18              // age is not equal to 18
```

1. Structure theorem is widely acknowledged to have first been introduced in Böhm, C., and Jacopini, G., "Flow diagrams, turing machines and languages with only two formation rules", in Communications of the ACM, Volume 9(5), May 1966, pp366-371.

Alternatively, relational operators can be used to compare to values to determine whether one value is less than or greater than the other. There are four such operators: less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=). the following are examples uses of relational operators:

```
age < 18                // age is less than 18
age > 18                // age is greater than 18
age <= 18               // age is 18 or less / at most 18
age >= 18               // age is 18 or greater / at least 18
```

More complex conditions can also be expressed by combining several conditions together using ***logical operators***. We consider three logical operators: ***conditional AND*** (`&&`), ***conditional OR*** (`||`) and the ***logical negation*** (`!`) operators[2]. The syntax for these three operators are as follows:

condition1 `&&` *condition2*
condition1 `||` *condition2*
`!`*condition*

How these complex conditions are evaluated is most simply shown using ***truth tables***. Consider the truth table for the conditional AND (`&&`):

condition1	*condition2*	*condition1* `&&` *condition2*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

The above truth table shows that if both *condition1* and *condition2* joined by the conditional AND operator are true, then overall the condition is true. However, if either or both of *condition1* or *condition2* are false, the overall condition is false.

The truth table for the conditional OR (`||`) is as follows:

condition1	*condition2*	*condition1* `||` *condition2*
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

From the truth table, you can see that if either or both *condition1* or *condition2* are true, the overall condition is true. The overall condition only evaluates to false if both *condition1* and *condition2* are false.

2. There are also the boolean logical AND (`&`), boolean logical OR (`|`), and the boolean logical XOR (`^`) operators which we do not consider in this unit. Use of these operators is usually restricted to more advanced programming applications and are not needed.

When working with the conditional AND or the conditional OR operators, it is also critical to understand the concepts of ***short-circuit evaluation***. If you reexamine the truth table for the conditional AND, you will notice that if *condition1* evaluates to false, there is no possibility that the overall condition will ever be true. Thus to evaluate *condition2* would be a waste of computation time. Similarly, if you reexamine the truth table for the conditional OR, if *condition1* evaluates to true, the overall condition must always be true. Thus, evaluating *condition2* would be a waste of computation time. This is precisely how the compiler works, and is known as short-circuit evaluation.

Finally, the truth table for the logical negation (`!`) operator is as follows:

condition	`!`*condition*
`true`	`false`
`false`	`true`

From this truth table, it can be seen that the logical negation operator just switches the result from true to false and vice versa.

2.2.2. Selection Control Structures

Applications need to make decisions as part of their functionality. For example, consider almost any Windows application where you can click on the File menu and then select the Open option, displaying the open file dialog. When that dialog is closed, the application makes several decisions based on your input: did you click Open or Cancel? If you clicked Open, does the specified file exist? Do you have permission to open that file? and so on.

To make these decisions, we consider two ***selection control structures***: the `if` statement, and the `switch` statement. These two control structures allow us to (i) test one or more conditions and execute some code depending on the outcome (the `if` statement), or (ii) to execute different code depending on the value of a particular variable/result (the `switch` statement).

The if and if-else Statement

The `if` statement allows us to test for one or more conditions within our program. Consider the question “is the sky blue?” On a clear day the answer would usually be yes (true), however on a cloudy day the answer would be no (false). Inserting the keyword ‘if’ in the question and adding an action to be performed if the answer is true, e.g., “if the sky is blue, children can go outside and play,” represents the behaviour of the `if` statement.

The basic syntax of the if statement in the C# programming language is as follows:

```
if(condition)
      true_statement;
```

The *true_statement* represents a statement executed if, and only if, the stated *condition* evaluates to true. The specified *condition* follows the rules outlined in Section 2.2.1. If more than one statement is required, then a code block can be used to group statements together, as follows:

```
/*************************************************************
** File: AgeChecker.cs
** Author/s: Justin Rough
** Description:
**     A simple program demonstrating how to use if
**     statements in the C# programming language.
*************************************************************/
using System;

namespace AgeChecker
{
    class AgeChecker
    {
        static void Main(string[] args)
        {
            Console.Write("How old are you? ");
            int age = Convert.ToInt32(Console.ReadLine());

            if (age == 18)
                Console.WriteLine("You are exactly 18 years old!");
            if (age != 18)
                Console.WriteLine("You are not 18 years of age.");
            if (age < 18)
                Console.WriteLine("You are not yet 18 years old.");
            if (age > 18)
                Console.WriteLine("You've been 18 for at least a year.");
            if (age <= 18)
                Console.WriteLine("You are at most 18 years old.");
            if (age >= 18)
                Console.WriteLine("You are at least 18 years old.");
        }
    }
}
```

Figure 2.1: Example of if statements and conditions

```
if(condition)
{
    true_statement1;
    true_statement2;
    ...
}
```

A simple example of using `if` statements is shown in Figure 2.1. The example prompts for and reads the user's age and then performs a series of `if` statements to display several messages appropriate for the age entered.

Closely related to the `if` statement is the `if-else` statement that additionally specifies one or more statements to be executed if the stated condition evaluates to false by using the else keyword, as follows:

```
if(condition)
    true_statement;
else
    false_statement;
```

Note that either or both of the *true_statement* and/or *false_statement* can be replaced with a code block if required.

Like all control structures, it is also possible to nest `if`/`if-else` statements, i.e., an `if`/`if-else` statement that appears inside another `if`/`if-else` statement. This is relatively straightforward, where the second `if` statement can appear in the *true_statement* section:

```
if(condition1)
    if(condition2)
        true_statement;
    else
        false_statement1;
else
    false_statement2;
```

or it could appear in the *false_statement* section:

```
if(condition1)
    true_statement1;
else
    if(condition2)
        true_statement2;
    else
        false_statement;
```

or it could appear in both sections:

```
if(condition1)
    if(condition2)
        true_statement1;
    else
        false_statement1;
else
    if(condition3)
        true_statement2;
    else
        false_statement2;
```

Note that there is no limit on the levels of nesting, e.g., you can have an `if`/`if-else` statement inside an `if`/`if-else` statement, inside an `if`/`if-else` statement, inside an `if`/`if-else` statement, and so on.

Even though these are simple extensions to what we have learned about `if`/`if-else` statements, there are two problems that can occur. First, consider the following two examples:

```
if(condition1)                          if(condition1)
    if(condition2)                          if(condition2)
        true_statement;       and               true_statement;
    else                                else
        false_statement;                    false_statement;
```

At first glance, the left example appears to be an `if-else` nested in an `if` statement and the right example appears to be an `if` statement nested in an `if-else` statement. However recall from Section 1.2 that the use of whitespace (spaces, tabs, and line breaks) does not alter the meaning of a statement, thus the two statements above are functionally equivalent. Although they appear to have a different logical structure, the compiler will interpret them as the same. This is known as the ***dangling else*** problem, where the interpretation of the statement is ambiguous, i.e., which `if` statement does the `else` belong to? This problem is very difficult to diagnose/debug because it is hard to spot – the code looks correct in each case. In C#, the compiler will always interpret the above `if` statement as shown on the left.

The dangling else problem is easily solved by using code blocks, allowing the programmer to explicitly specify the intended semantics, as follows:

```
if(condition1)                              if(condition1)
{                                           {
    if(condition2)                              if(condition2)
        true_statement;          and                true_statement;
    else                                    }
        false_statement;                    else
}                                               false_statement;
```

Another problem occurs when a sequence of `if` statements are used to handle multiple ranges or values, as follows:

```
if(condition1)
    true_statement1;
else
    if(condition2)
        true_statement2;
    else
        if(condition3)
            true_statement3;
        else
            if(condition4)
                true_statement4;
            else
                false_statement;
```

Such sequences of `if` statements are used quite regularly in programming and quickly become difficult to read as the `if` statements continue to tab across the screen. When this structure occurs, we change the indentation rules slightly and treat "`else if`" as a special case[3], as follows:

```
if(condition1)
    true_statement1;
else if(condition2)
    true_statement2;
else if(condition3)
    true_statement3;
else if(condition4)
    true_statement4;
else
    false_statement;
```

3. Note that some languages, e.g., Perl, actually have a specific language support for this structure (Perl has *if*, *else*, and *elsif* keywords).

Task 2.1. Personal Income Tax (Required)

Objective: The concepts of conditions and if statements are fundamental to the development of most applications. This task requires you to combine these skills in the development of a meaningful application.

According to the Australian Taxation Office web site[4] the taxation rates for the 2011-12 financial year are as follows:

Taxable income	Tax on this income
\$0 – \$6,000	Nil
\$6,001 – \$37,000	15c for each \$1 over \$6,000
\$37,001 – \$80,000	\$4,650 plus 30c for each \$1 over \$37,000
\$80,001 – \$180,000	\$17,550 plus 37c for each \$1 over \$80,000
\$180,001 and over	\$54,550 plus 45c for each \$1 over \$180,000

Your task is to write an application that determines the tax payable for any taxable income according to the rules in this table.

The switch statement

In the last sub-section we saw how a large number `if-else` statements used together can result in code that is both complex and difficult to read. When such a series of `if-else` statements is required to test for distinct values, a `switch` statement is the preferred structure to use instead[5]. The `switch` statement compares the result of an expression with one or more values, performing a different set of statements depending on which value was matched. The syntax for the `switch` statement is as follows:

```
switch(expression)
{
case value1:
      [statement;
      [...]
      break;]
[case value2:
      [statement;
      [...]
      break;]
[...]]
```

4. http://www.ato.gov.au
5. Note however that the switch statement cannot be used to test the value of more than one variable and cannot be used to test ranges of values.

```
[default:
      [statement;
      [...]
      break;]]
[...]
}
```

In the above syntax, the *expression* is first evaluated to determine the actual value to be tested, e.g., by completing any method calls to determine their return value, and then the result is compared against the values indicated in the different `case` statements (*value1* and *value2* in the above example). If a matching value is found, the statements appearing under that `case` are executed. If no match is found, i.e., the result does not match any specific `case`, then the statements appearing in the `default` section, if present, are executed instead. The statements appearing under a `case`/`default` section are executed up until the `break` statement, at which point the switch statement is complete and execution continues with the next statement appearing after the switch statement. Note that if the result does not match any case and no default section is present, no statements are executed as part of the switch statement and execution continues with the next statement appearing after the switch statement.

It is also important to understand that the syntax for the switch statement also makes it possible to have a single set of statements used for multiple `case` sections by skipping the statement/s and `break` statement for one or more cases, i.e.,

```
case value1:
case value2:
case value3:
      statement;
      ...
      break;
```

In the above example, the same statement/s would be executed if *expression* evaluated to any of *value1*, *value2*, or *value3*.

An example switch statement is shown in Figure 2.2, where the user is prompted for what grade they achieved in a unit, and the system then gives them a message appropriate for the grade they achieved. If their grade is not known to the system, this is picked up by the `default` case and a message is displayed indicating an unknown grade.

The conditional operator (?:)

The C# programming language also provides a ternary operator, known as the conditional operator, which provides a kind of short-hand for trivial if statements. The conditional operator uses the following syntax:

condition `?` *true_part* `:` *false_part*

To see the conditional operator in action, consider the following code:

```
Console.Write("How many students can you see? ");
int numStudents = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("You can see {0} student{1}.", numStudents,
         (numStudents == 1 ? "" : "s"));
```

```
/*********************************************************
** File: Grader.cs
** Author/s: Justin Rough
** Description:
**     A simple program demonstrating how to use switch
**     statements to check multiple possible results for an
**     expression and perform different statements.
*********************************************************/
using System;

namespace Grader
{
    class Grader
    {
        static void Main(string[] args)
        {
            Console.Write("What grade did you achieve? ");
            string grade = Console.ReadLine();

            switch (grade.ToUpper())
            {
                case "HD":
                    Console.WriteLine("You got >= 80 marks, great!");
                    break;
                case "D":
                    Console.WriteLine("You got >= 70 marks, well done!");
                    break;
                case "C":
                    Console.WriteLine("You got >= 60 marks, good!");
                    break;
                case "P":
                    Console.WriteLine("You got >= 50 marks, not bad!");
                    break;
                case "N":
                    Console.WriteLine("You didn't pass, try again!");
                    break;
                default:
                    Console.WriteLine("Sorry, I don't know that grade.");
                    break;
            }
        }
    }
}
```

Figure 2.2: Example of a switch statement

The last line of the code uses the conditional operator to decide whether or not to output the letter 's' after the word 'student' – if the number of students entered is one, then the letter 's' is not output, otherwise it is, resulting in the following output examples:

```
You can see 0 students.
You can see 1 student.
You can see 2 students.
...
```

The same result could have been achieved by replacing the last `WriteLine` statement with the following code:

```
if(numStudents == 1)
     Console.WriteLine("You can see 1 student.");
else
     Console.WriteLine("You can see {0} students.", numStudents);
```

The conditional operator is commonly used by more advanced programmers to save writing a lot of code with a trivial condition, such as in the example above. However, some people have difficulty learning how to read and write conditional expressions, and there is no specific advantage to using conditional expressions other than to save typing, thus we do not examine the conditional operator further in this unit. Feel free to experiment with and use the conditional operator in your work however.

2.2.3. Repetition Control Structures

Loops are also critical to most applications, and there are many examples where loops are needed: keep executing a computation until the solution is found, keep printing until all pages are output, keep reading data until the end of the file, and so on. Applications that run in a windowed environment rely on loops for their operation – they all have a loop known as the event loop which retrieves the next event from the operating system (mouse click, key press, and so on), which is then "dispatched" to an appropriate event handler for processing. Similarly, many games use loops to keep the game world running, e.g., process user input, character animations, NPC actions/AI, object movement/physics, queueing/buffering more music/sound, and so on.

Generally, loops consist of a loop statement and the ***loop body***. The loop statement is how we express to the compiler how long/how many times we want the loop to execute. The loop body is the statement or statements that are executed for each step of the loop. A key term that is used when discussing loops is ***iteration***, which has two meanings. The word iteration can refer to either (i) the general act of repetition using a loop, but also (ii) a single repetition/step in a loop, i.e., if a loop runs five times there are five iterations of the loop.

In this section we examine three ***repetition control structures***: the `while` loop, the `do-while` loop, and the `for` loop. The `foreach` loop is examined later in Section 3.2.3. These three control structures allow us to (i) execute some code as long as a condition is true (the `while` loop), (ii) execute some code until a condition is no longer true (the `do-while` loop), and (iii) execute some code for a number of times (the `for` loop). Finally, we complete our study of looping structures by examining the `break` and `continue` keywords

The while loop

The `while` loop allows us to execute some code as long as a condition evaluates to `true`. The syntax for the while loop is as follows:

```
while(condition)
        statement;
```

If more than one statement is required, then a code block can be used:

```
while(condition)
{
        statement 1;
        statement 2;
        ...
}
```

To understanding the operation of the `while` loop properly, you must understand how the condition is evaluated. Upon reaching a `while` loop, the program will operate as follows:

i. Evaluate the condition, if false then leave the `while` loop;
ii. Perform the statements in the loop (the loop body); and
iii. Return to step (i);

Consider the example shown in Figure 2.3 which shows how to use a `while` loop (along with a `switch` statement) to present a simple menu to the user. Although we can improve this code slightly after we have gained more knowledge of programming and/or C#, the fundamental logic shown in this example will remain the same, and it is important that you learn it.

The program begins by showing the user the menu of options and obtaining their selection. When the program execution reaches the `while` loop, the condition is immediately evaluated, which states that the loop should only run if the user did not select option `0` (the exit option). Thus, if the user selected the exit option first, the program would now terminate. If the user did not select option `0`, then the loop body is executed. There are three possibilities: the user selected option `1` (say hello), the user selected option `2` (say goodbye), or the user chose an option that does not exist. These possibilities are handled effectively by a `switch` statement, which we studied in Section 2.2.2. After processing the user's menu selection, the menu is displayed again and a new selection is obtained. The loop now continues by evaluating the condition once again, and only continues if the user did not select option `0` (the exit option).

The do-while loop

The `do-while` loop is almost identical to the `while` loop, except for one difference. The syntax for the `do-while` is as follows:

```
do
    statement;
while(condition);
```

or with a code block for multiple statements:

```
do
{
    statement 1;
    statement 2;
    ...
} while(condition);
```

Immediately you will notice that the condition has moved to the bottom of the loop. Unlike a `while` loop, where the condition is evaluated before the loop body is executed, in a `do-while` loop the condition is evaluated after the loop body has been executed. Moreover, the `do-while` loop operates as follows:

i. Perform the statements in the loop;
ii. Evaluate the condition, if false then leave the `while` loop; and
iii. Return to step (i);

Note that compared to the operation of the `while` loop, steps (i) and (ii) have been reversed. This has the important outcome that the loop body of a `do-while` will always be executed at least once, given that the condition will only be evaluated for the first time only after the loop body has been executed once.

```
/**********************************************************
** File: SimpleMenu.cs
** Author/s: Justin Rough
** Description:
**     A simple program that demonstrates how to use a while
**     loop to construct a simple menu.
**********************************************************/
using System;

namespace SimpleMenu
{
    class SimpleMenu
    {
        static void Main(string[] args)
        {
            Console.WriteLine("1. Say hello");
            Console.WriteLine("2. Say goodbye");
            Console.WriteLine("0. Exit");
            Console.Write("Selection? ");
            int selection = Convert.ToInt32(Console.ReadLine());
            while (selection != 0)
            {
                switch (selection)
                {
                    case 1:
                        Console.WriteLine("Hello!");
                        break;
                    case 2:
                        Console.WriteLine("Goodbye!");
                        break;
                    default:
                        Console.WriteLine("Invalid option!");
                        break;
                }
                Console.WriteLine("1. Say hello");
                Console.WriteLine("2. Say goodbye");
                Console.WriteLine("0. Exit");
                Console.Write("Selection? ");
                selection = Convert.ToInt32(Console.ReadLine());
            }
        }
    }
}
```

Figure 2.3: Example of a while loop

The for loop

The `for` loop is used when there is a need to perform one or more statements a pre-defined number of times. The syntax for the `for` loop is as follows:

```
for(initialisation ; condition ; increment)
    statement;
```

or with the use of a code block for multiple statements:

```
/**********************************************************
** File: TimesTable.cs
** Author/s: Justin Rough
** Description:
**      A simple program that demonstrates how to use for loops
** to produce a multiplication table for the input 1-12.
**********************************************************/
using System;

namespace TimesTable
{
    class TimesTable
    {
        static void Main(string[] args)
        {
            for (int row = 1; row <= 12; row++)
            {
                for (int col = 1; col <= 12; col++)
                {
                    Console.Write("{0,3} ", row * col);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Figure 2.4: Example of a for loop

```
for(initialisation ; condition ; increment)
{
    statement 1;
    statement 2;
    ...
}
```

There are three components indicated after the `for` keyword:

- *initialisation* – a single statement that is executed before the loop begins;
- *condition* – the condition under which the loop runs (same behaviour as the condition in a `while` loop); and
- *increment* – a single statement that is executed at the end of each iteration of the loop[6].

An example of using `for` loops is shown in Figure 2.4. This example shows how to use two `for` loops to produce a simple multiplication table. Two loops are used, one which has an iteration for each row of the multiplication table (using variable `row`), and another which has an iteration for each column for a single row (using variable `col`). Note the use of composite formatting in the `WriteLine` statement, examined in Section 2.9.

What may not be immediately apparent when studying a `for` loop in C#, is that it is actually very similar to a `while` loop. In fact, any `for` loop can be converted to a `while` loop, and vice-versa, using the following structure/syntax:

6. Although the word increment is used here, and most of the time an increment statement would be used, the statement can in fact be anything.

```
initialisation;
while(condition)
{
    statement 1;
    statement 2;
    ...
    increment;
}
```

Note that the three components in a `for` statement are still present: *initialisation* appears before the loop, the *condition* appears in the `while` statement, and the *increment* appears before the end of the loop body. Although the `for` and `while` are completely interchangeable, `for` loops are generally used for counting and `while` loops for non-counting loops.

break and continue

Finally, the C# programming language includes two special statements that provide additional control over repetition control structures: the `break` statement and the `continue` statement.

The `break` statement, which we have already seen in the context of the `switch` statement, allows us to "break out" of a loop, terminating its execution. Program execution will continue at the first statement appearing immediately after the loop when a `break` statement is executed.

The `continue` statement immediately finishes the current iteration, and continues with the next iteration of the loop. Program execution continues with the increment statement (if a `for` loop is used), then the condition is re-evaluated, and if true, the loop body is executed again.

A simple example is shown in Figure 2.5. Ignoring the two `if` statements, we see a single loop that would print out the numbers from `1` to `100`. However if we add in the first `if` statement, the loop is broken if the value of `i` exceeds `10`, i.e., the loop now only prints the numbers from `1` to `10`. Adding the second `if` statement checks that if the remainder of `i / 2` is `1`, i.e., an odd number, we should immediately continue to the next iteration. Thus, the loop actually prints out the even numbers appearing between `1` and `10`.

Task 2.2. Finding Factors (Required)

Objective: The ability to apply loops to solve a problem is a critical skill for any software developer. This task exploits loops to solve a simple mathematics problem.

The factors of a number are any values that evenly divide the number. For example, the number 42 can be evenly divided by the values 1, 2, 3, 6, 7, 14, 21, and 42. Your task is to write a program that displays all of the factors for a number. Try to write your program so that it produces output similar to the following:

```
Please enter a number: 42
1 * 42 = 42
2 * 21 = 42
3 * 14 = 42
6 * 7 = 42
```

Hint: If a value *a* is divisible by *b*, then *a* `%` *b* `== 0`

```
/***********************************************************
** File: EvenNumbers.cs
** Author/s: Justin Rough
** Description:
**     A simple program that demonstrates how to use the break
** and continue statements in a loop to control its execution.
***********************************************************/
using System;

namespace EvenNumbers
{
    class EvenNumbers
    {
        static void Main(string[] args)
        {
            for (int i = 1; i <= 100; i++)
            {
                if (i > 10)
                    break;
                if (i % 2 == 1)
                    continue;
                Console.WriteLine(i);
            }
        }
    }
}
```

Figure 2.5: Example loop using break and continue

2.3. Defining Classes

In Session 1, we identified the structure of object-oriented applications as consisting of a collection of interacting objects. If we consider these objects closely, we can see that some objects are very similar in their nature.

For example, if you were developing an application for the Deakin library to manage its borrowings, the clients (whoever borrows the books) could include undergraduate students, post-graduate students, academic staff members, administrative staff members, technical support staff members, and so on. Each student or staff member would be represented in the application by a separate object, however these objects would be very similar. For example, each student/staff member will have a family name and given name, an ID number, an address, and a contact telephone number. These objects are said to belong to a single classification (a class), and that the objects are instances of that class.

When writing the code for an object-oriented application, we are undertaking the task of writing one or more classes. We then instantiate these classes to create objects that belong to a particular class. The C# syntax for defining a class is as follows:

```
[access_modifier ]class class_name
{
    [access_modifier ]class_member
    ...
}
```

There are three elements in the above syntax that can change. The *class_name* field is the name given to the class, which must follow the same rules as variable names (see Section 1.4). Like a variable name, the name of the class should be representative of the data that it will represent or the role that it will play in the application. For the library book borrowings example, we might name our class Person, Client, Member, or Borrower to represent the people who may borrow books from the library.

The *class_member* field represents any number of operations, attributes, or other elements that are defined as members of the class, i.e., part of the class. Defining class members is addressed in the next sections.

The ***access modifier*** (*access_modifier* field) optionally appears on both the class and separately for each member, and determines what other parts of your application are permitted to access that class/class member. This is important because, for example, if access is permitted to an operation, the code that has access to that operation can invoke/call that operation. The use of different access modifiers allows us to implement the abstraction and encapsulation concepts introduced in Session 1, as discussed below.

There are five possible access modifiers: `public`, `private`, `internal`, `protected`, and `protected internal`. We will examine `protected` and `protected internal` access modifiers as part of our study of inheritance in Section 5.3. The `public` and `private` access levels are the easiest to understand. A class/class member that has a `public` access modifier can be accessed by code anywhere in the program. A `private` class/class member however can only be accessed by code written as part of the same class. As such, those classes/class members marked as `public` form part of the abstraction presented by an object, i.e., they define the interface to an object. Those classes/class members marked as `private` are encapsulated (hidden) and are not accessible to other code.

The final access modifier, `internal`, is not used as often and makes a class/class member accessible to all of the code within a single ***assembly***. For the work we complete in this unit, each separate application that we will write is one assembly. Thus for our purposes, the `internal` access modifier is equivalent to the `public` access modifier. More advanced applications can consist of multiple assemblies, most commonly throught the use of Dynamic Link Libraries (DLLs). DLLs contain code (and potentially other resources) that can be used by an application at run time. Each DLL used by an application forms a separate assembly. If we were writing a class that is part of a DLL, the use of the `internal` access modifier would mean that all code inside the DLL could access the class/class member, but the applications using that DLL could not access that same class/class member.

As noted above, the access modifier is optional for both the class and for class members. If there is no access modifier indicated on a class, the compiler will treat the class as though it had an `internal` access modifier specified, i.e., classes have a default access modifier of `internal`. However class members missing an access modifier are treated as though they have a `private` access modifier specified, i.e., class members have a default access modifier of `private`. Thus by default, all classes are accessible by all code within a single assembly (our applications), and class members are only accessible by code within the same class, unless an access modifier is explicitly provided.

Importantly, we have already been defining classes in our C# programs. In fact, it is not possible to write a C# application without defining at least one class. If you review Figure 1.1 from Session 1, you will notice that we defined a class called `FirstProgram` that contained only a `Main` method. Both the `FirstProgram` class and the `Main` method did not have an access modifier specified, thus the `FirstProgram` class is actually internal, and the `Main` method is actually private.

Once a class has been defined, we can create objects using that class as a template. The syntax for creating an object from a class is as follows:

class_name variable_name;
variable_name = new *class_name*(*[parameters]*);

Alternatively, the above two lines can be combined together, as follows:

class_name variable_name = new *class_name*(*[parameters]*);

The syntax here is straightforward except for parameters, which are discussed in the context of constructors in Section 2.4. For example, to create an object named `jSmith` from a class `Person`:

```
Person jSmith = new Person();
```

2.4. Defining Class Operations

Recall from Session 1 that operations define the behaviour of an object, i.e., how an object acts and reacts. Operations are implemented in C# using ***instance methods***[7]. Instance methods usually either query (read) or manipulate (write) the state of an object. Examples:

- A light switch may have a method to query whether the switch is currently on or off, and a method that turns the light switch on or off;
- A keyboard may have a method to query whether a particular key is currently depressed or not, and a method that turns the caps lock/scroll lock/number lock lights on and off;
- A queue may have a method to query how many elements are currently stored in the queue (such as how many print jobs in a printer queue), and a method to enqueue an additional element in the queue (add a document to the queue for printing); and
- An avatar may have a method to query the current location (x, y, z location) of the avatar in the game world, and a method to record damage against the avatar.

There are three elements that must be considered when writing any method: input, output, and processing. The input to a method represents information that is required to complete the processing of the method. The output of a method is the results of the method, which could be some data retrieved, a calculation performed, or the success/fail of some request. The processing represents the actual steps taken to determine the output from the provided input.

For example, consider calculating the area of a rectangle. Mathematics tells us that the area is the product of the length and width of the rectangle. Thus for this example, we have the following elements:

- Input: Length, Width
- Output: Area
- Processing: Area = Length * Width

Input to a method can be from two sources: parameters to the method (input parameters), or read from the state (attributes) of an object. Parameters represent data passed into a method that contain data required by that method. Parameters are specific to a method, i.e., the parameters used by one method have no relevance to other methods, moreover one methods has no knowledge of the parameters used by other methods. Attributes store information that is relevant to the whole object, not just to a specific method. Attributes are examined in detail in Section 2.5, below.

7. Methods are also known as functions and the two terms are interchangeable. The term method has been used throughout this workbook however to match the prescribed textbook for this unit. Other textbooks and materials on the web will often use the term function instead.

Output from a method can be in three forms: parameters to the method (reference or output parameters, see Section 2.6), written to the state (attributes, see Section 2.5) of an object, or as a result returned by the method. Results that are returned by a method are known as their ***return value***. A return value is some data, usually of a simple data type, that indicates the outcome of a method. For example, a boolean (true/false) value could be returned to indicate the success or failure of an operation performed by a method, a numeric value could be returned to indicate how many records were affected/changed by the functionality in a method, a string could be returned indicating an error message, and so on.

The C# syntax for defining a method is as follows:

```
[access_modifier ]return_type method_name([parameter[, ...]])
{
      method_body
}
```

The *access_modifier* field was discussed previously in Section 2.3. The *return_type* field specifies the type of data used for a return value, or the keyword `void` that indicates no result is returned. The *method_name* must follow the same rules as a variable name (Section 1.4). Zero or more parameters can be indicated inside the parenthesis (‘( )’), known as the ***argument list*** (or parameter list), which are the same as a variable declaration except they are separated by commas and do not finish with a semi-colon (‘;’).

Calling a method then changes slightly depending on whether you are calling the method from another method in the same class:

```
method_name([parameter[, ...]]);
or
this.method_name([parameter[, ...]]);
```

or from a method defined in another class through an object:

```
object_name.method_name([parameter[, ...]]);
```

An example is provided in Figure 2.6[8] which, although not very realistic, effectively demonstrates the concepts for defining classes, defining instance methods and invoking instance methods discussed so far. Note that the code in the example is not complete, only the relevant code is shown here. A class `TrivialClass` is defined containing a single public method named `GetMessage()`. This method has no parameters and returns a string containing the simple message `Hello Object-Oriented Development!` The `Main` method demonstrates how to create an object from this class, and then how to invoke the `GetMessage()` method to retrieve the message which is then displayed on the console.

Constructors

Constructors are a special type of method that is invoked when an object is first created, used to prepare/initialise the object for use. We examine constructors in detail in Section 4.4, however introduce the basic concepts here as they are very useful for initialising objects. The syntax for a constructor is the same as for other methods, except (i) no return type is indicated, (ii) the name of the method matches the name of the class, and (iii) is almost always public, i.e.,

8. The code in the figure is only an extract from the full code which is available in DSO.

```
class TrivialClass
{
    public string GetMessage()
    {
        return "Hello Object-Oriented Development!";
    }
}

class Program
{
    static void Main(string[] args)
    {
        TrivialClass trivialObject = new TrivialClass();
        Console.WriteLine(trivialObject.GetMessage());
    }
}
```

Figure 2.6: Trivial class definition, object creation, and method invocation example

```
[access_modifier ]class class_name
{
    [access_modifier ]class_name([parameter[, ...]])
    {
        method_body
    }
    ...
}
```

Constructors are not invoked like other methods, and are instead invoked as part of object creation. Recall the syntax for object creation shown in Section 2.3 (combined version):

```
class_name variable_name = new class_name([parameter[, ...]]);
```

Here, the parameters in the object creation statement must match the argument list defined for the constructor. A simple class demonstrating the use of constructors is shown in Figure 2.7[9]. This class continues our earlier example of a rectangle and includes the following features:

- Two instance variables (see Section 2.5) to hold the length and width of the rectangle;
- A constructor which will initialise these values during object creation; and
- Two instance methods to calculate and return the area and perimeter of the rectangle.

2.5. Defining Class Attributes

To implement attributes, we define ***instance variables***[10]. Instance variables are no different to the variables discussed previously in Section 1.4, except instead of being part of a method, instance variables are part of an object, i.e., an instance. These variables are accessible by all methods defined in the same class, and potentially by methods defined outside of that class (dependent upon the access modifier used). The syntax for declaring an instance variable is almost identical to the syntax we have already seen for declaring variables. In particular, the only change is the addition of an optional access modifier:

```
[access_modifier ]type name1[ = value][, name2[ = value][, ...]];
```

9. Complete source code for this example can be found in DSO.
10. Instance variables are often just referred to as attributes. However in this workbook we use different terms to differentiate clearly between the general object-oriented concept (attribute) and the language specific mechanism (instance variable).

```
class Rectangle
{
     private double _Length;
     private double _Width;

     public Rectangle(double length, double width)
     {
          _Length = length;
          _Width = width;
     }

     public double GetArea()
     {
          return _Length * _Width;
     }
}
```

Figure 2.7: Simple example of a constructor to initialise an object

Instance variables should almost always be declared `private` to preserve encapsulation – recall from Session 1 that encapsulation maintains a separation between the external view/ interface of an object and its implementation. Instance variables represent the implementation of an attribute, i.e., how the data is actually stored, thus should be hidden outside of the class they are defined in. Other (public/protected/internal) class members should then be defined to provide the interface to the instance variables. Consider how a date is stored in the following variable declarations:

```
int date = 20120301;

string date = "March 1st, 2012";

byte day = 1;
byte month = 3;
ushort year = 2012;
```

Each of the above declarations store the same date, but represent that date differently. Each representation has its advantages, e.g., the first representation is useful for sorting dates, the second is in a pre-formatted user-friendly form (for displaying on the console/in a GUI label), and the third is useful if data would regularly be searched for/categorised by month and/or by year. Thus, different implementations are possible for the same data.

Importantly, throughout the lifetime of an application the way data is represented may need to be changed. For example, introducing new functionality to an application such as network or database support may change which representation is the most appropriate/efficient for some data. If the instance variable storing that data was directly accessible by methods in other classes, changing the representation of that data would then require changes to be made to all classes that access that instance variable to handle the new representation. By keeping instance variables `private` and providing a separate (`public`) interface, it is possible to modify the representation of the data while maintaining the same interface, limiting the change to the one class. This approach has the additional advantages of improving the reusability of the class and reducing development time.

Providing an interface to instance variables is achieved using either accessor methods or properties, which we examine below. First however, it is necessary to consider how to name these elements. Instance variables, accessor methods, mutator methods, and properties will all have very similar names as they are all referring to the same data, e.g., for a Person class, they could all refer to a person's given name. Given that instance variables are usually hidden/ private, it is common to give them a prefix of some sort, e.g., for a given name we could use:

```
string _GivenName;
string fGivenName;
string m_GivenName;
```

The examples above represent three different prefixes ('`_`', '`f`', and '`m_`') that are often given to instance variables, although others exist, and they tend to change over time. Examples in this unit will use the single underscore ('`_`') which seems to be the prefix favoured by most developers today[11]. In the above prefixes, the '`f`' is often interpreted as an abbreviation for 'field' (another name for an instance variable), and '`m_`' is usually interpreted as 'member' (as in class member), although not strictly.

Accessors and mutators methods

Accessor methods and ***mutator methods***, sometimes referred to collectively as accessor methods, are short methods defined to provide the public interface to the instance variables in a class. Accessor methods provide the functionality for reading a value and usually begin with the prefix `Get`. Mutator methods provide the functionality for writing a value and usually begin with the prefix `Set`. An example of an instance variable, accessor and mutator methods is shown in Figure 2.8 for an attribute for a given name. Note that in this example, the accessor method only returns the value of the instance variable containing the given name, however in more complicated examples accessor methods can also perform some basic calculations, e.g., to obtain the current total of an invoice, to retrieve credit card information from a database, and so on.

Properties

One of the disadvantages of using accessor and mutator methods is that if only a simple value as being read/written to it is not as intuitive as accessing a variable. For example:

```
sales.SetCount(sales.GetCount() + 1);
```

is not as intuitive/obvious/easy to read as

```
sales.Count = sales.Count + 1;
```

For this purpose many modern object-oriented programming languages, including C#, introduce the concept of a ***property***. Properties embed the concepts of accessor and mutator

```
private string _GivenName;

public string GetGivenName()
{
     return _GivenName;
}

public void SetGivenName(string value)
{
     _GivenName = value;
}
```

Figure 2.8: Accessor and mutator for a given name attribute

11. Note that you are not expected to adopt a specific prefix, however you should apply one of these prefixes consistently throughout your code.

```
private string _GivenName;
public string GivenName
{
     get
     {
          return _GivenName;
     }
     set
     {
          _GivenName = value;
     }
}
```

Figure 2.9: Property for a given name attribute

methods into the language syntax itself, allowing accessor and mutator methods to be used while maintaining a syntax similar to accessing a variable/attribute. Properties are defined in C# using the following syntax:

```
[access_modifier ]type name
{
     [[access_modifier ]get
     {
          accessor_body
     }]
     [[access_modifier ]set
     {
          mutator_body
     }]
}
```

In the syntax above there are two code blocks shown, one labelled with the `get` keyword and the other labelled with the `set` keyword[12]. As suggested in the syntax, above, the `get` code block represents the accessor code (*accessor_body*), and the `set` code block represents the mutator code (*mutator_body*). Although both code blocks are shown as optional, at least one must be provided. By using different combinations of these code blocks, this results in three possible types of properties:

- A read/write property – both `get` and `set` blocks are defined;
- A read-only property[13] – only the `get` block is defined; and
- A write-only property – only the `set` block is defined.

Note that the access modifiers next to the `get` and `set` keywords are addressed in the next section. What is not clear from the above syntax is how data is input and output from the `get` and `set` blocks. Just like an accessor method, the `get` block uses the `return` keyword to return the data value. The input to the `set` block, i.e., the new value for the attribute, is provided by the keyword `value`. The data type of both the data returned by the `get` block and the `value` provided to the `set` block match the *type* of the property. An example of a property for a given name for a class Person is shown in Figure 2.9, which is equivalent to the previous example in Figure 2.8 which used accessor and mutator methods.

12. Some C# textbooks refer to these as the get accessor and set accessor, respectively.
13. Note that the C# programming language also defines the `readonly` keyword which has nothing to do with a read-only property as discussed here.

```
public string GivenName { get; set; }
```

Figure 2.10: Auto-implemented property for a given name attribute

Note that although properties offer equivalent functionality to accessor and mutator methods that is more intuitive, it is still important to know how to write accessor and mutator methods because not all programming languages support properties[14]. Accessor and mutator methods are the only choice in such languages.

Auto-implemented Properties

Auto-implemented properties were introduced in an updated specification of C# which appeared with the release of Visual Studio 2008. When an auto-implemented property is used, the C# programming language will automatically provide a matching attribute for that property, i.e., the programmer does not need to manually define the implementation of the property. This can save time during the development of an application while still allowing the implementation to be changed later while maintaining a consistent interface. An example of an auto-implemented property is shown in Figure 2.10, which is equivalent to the previously shown example of a property shown in Figure 2.9. In this example, the `get` and `set` code blocks are not defined, instead are replaced by a semi-colon. This instructs the compiler to provide the implementation.

Read/write, read-only, and write-only properties are also possible, but it does not make sense to only omit either the `get` or `set`, as this would effectively indicate that there is no possibility to ever retrieve or store (respectively) a value in the property. Instead, recall that access modifiers can be specified separately for the `get` and `set` sections. With the addition of a `private` access modifier it is possible to achieve the equivalent of a read-only or write-only property, e.g.,

```
public string ReadableText { get; private set; } // read-only property
public string WritableText { private get; set; } // write-only property
```

In these examples, the `ReadableText` property can only be read outside the class, but can also be modified from within the same class. Similarly, for the `WritableText` property, it is possible to modify the property from outside of the class, but it can only be read from within the same class.

An example class

Combining the elements we have seen in this chapter so far, we can now define a more complete example `Rectangle` class, as follows:

```
public class Rectangle
{
     private int _Length;
     public int Length
     {
          get { return _Length; }
          set { _Length = value; }
     }

     private int _Width;
```

14. Those students studying the Games Design and Development course will encounter this when learning the C++ programming language, which does not support properties.

```
        public int Width
        {
            get { return _Width; }
            set { _Width = value; }
        }

        public Rectangle(int length, int width)
        {
            _Length = length;
            _Width = width;
        }

        public int Perimeter()
        {
            return _Length * 2 + _Width * 2;
        }

        public int Area()
        {
            return _Length * _Width;
        }
    }
```

Task 2.3. Working with Classes I (Optional)

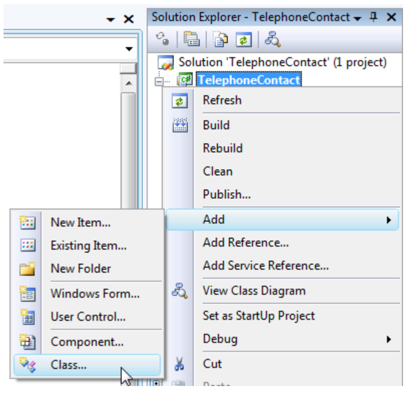
Objective: As the fundamental building block of an object-oriented application, developing classes is a crutial skill if you are to succeed. Thus this task leads you through the development of your first class to ensure you understand the basics. Follow the instructions below carefully to ensure you learn what is required to write class.

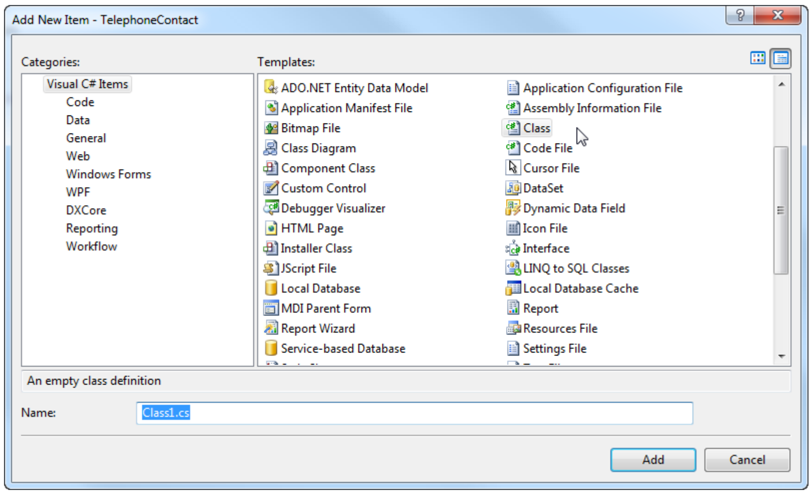
The task:

> You are required to write a class `Person` to store the basic contact information for a telephone book entry: family name, given name/initial, and telephone number. Each of these attributes is of type `string`. Attributes should be properly encapsulated and a constructor should be provided with parameters to initialise each of the three attributes. You are also required to write a `Main` method that creates a `Person` object, prompts the user for the required data, then displays it on the screen in an appropriate format.

The solution:

Start by creating a project in Visual Studio as we have seen previously. The solution name used in this example is `TelephoneContact`. Until now, we have only seen how to write programs containing a single class. In this project we will have two classes: one which contains the `Main` method (which is created automatically), and the `Person` class that we have been instructed to create. To add a new class, right click on your project in the **Solution Explorer** (the second line, highlighted in blue), click on the **Add** menu and then select the **Class...** option.





An empty class definition

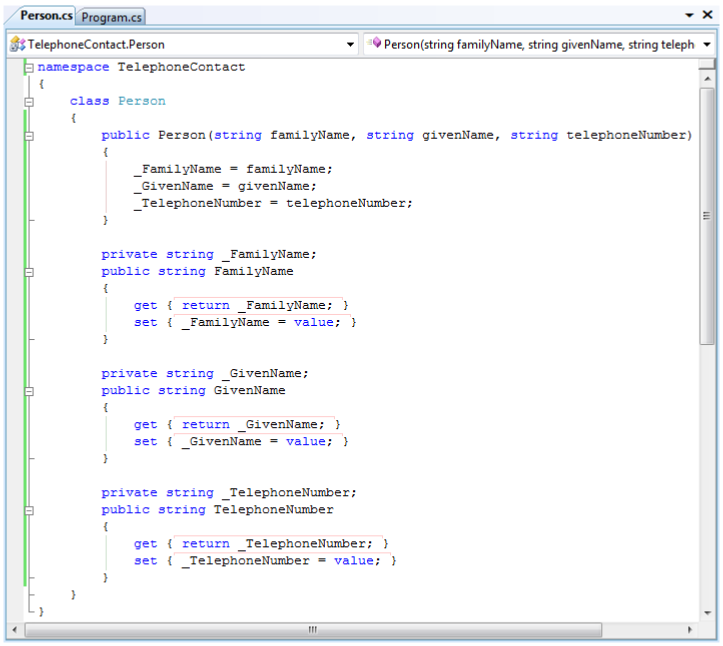
Name: Class1.cs

A dialog will appear where you select a template for the class and provide a file name. For this unit, we will only use the **Class** template (a simple class). Once selected, provide the `Person.cs` file name (the name of the class, Person, followed by the `.cs` extension) and click on the **Add** button or press the ENTER key on your keyboard.

This will create the skeleton for a class Person with no members defined as yet. To build our class, we need to perform the following steps:

- The problem tells us that there are three attributes: the family name, given name, and telephone number, and that these attributes are all of type string.. To create an attribute, we first need to create an instance variable to store the data. Create three instance variables matching these attributes. In this example we have called them `_FamilyName`, `_GivenName`, and `_TelephoneNumber`. As with all instance variables, these should all be private.
- To provide an interface, we now create three public properties to encapsulate the instance variables: `FamilyName`, `GivenName`, and `TelephoneNumber`.
- The only step remaining is to create the constructor, which is to have three parameters to initialise the attributes. What this means, is that we need three parameters to the method whose values will be used to initialise (set an initial value of) the instance variables. To differentiate the names of the parameters from the properties and attribute names already in the class, we use Camel Case in this example: `familyName`, `givenName`, and `telephoneNumber`. All that is required in the method body is three assignment statements to copy the values from the parameters to the instance variables.
- Finally, there is no need for any of the using statements added by Visual Studio by default, as we aren't using anything from the library, so we have removed those as well.

We have now completed our class, as shown in this screenshot.

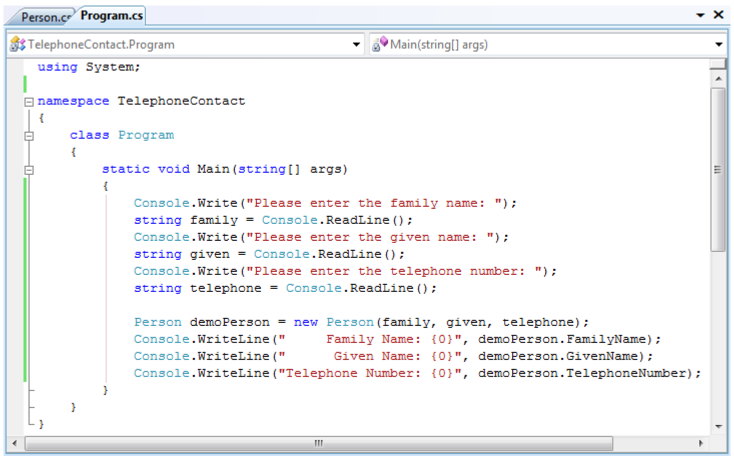


```
namespace TelephoneContact
{
    class Person
    {
        public Person(string familyName, string givenName, string telephoneNumber)
        {
            _FamilyName = familyName;
            _GivenName = givenName;
            _TelephoneNumber = telephoneNumber;
        }

        private string _FamilyName;
        public string FamilyName
        {
            get { return _FamilyName; }
            set { _FamilyName = value; }
        }

        private string _GivenName;
        public string GivenName
        {
            get { return _GivenName; }
            set { _GivenName = value; }
        }

        private string _TelephoneNumber;
        public string TelephoneNumber
        {
            get { return _TelephoneNumber; }
            set { _TelephoneNumber = value; }
        }
    }
}
```

The last stage of our development is to write the `Main` method itself:

- Before we can create our object we need the required data (family name, given name, and telephone number), so start by declaring three `string` variables to store this information then obtain them from the user.
- We can now create the object using the object creation syntax shown in Section 2.3, matching the three elements of data we obtained from the user to the parameters in the constructor function (family name first, followed by given name, and finally telephone number).
- Displaying the information from the object is no different to `Console.WriteLine` statements we have already seen but we use the properties in the objects instead of method variables.
- Remove the unnecessary using statements to clean up the code, leaving only the System namespace (which contains the `Console` object).

The result is the code shown in the screen shot, and we have finished our task.



```
using System;

namespace TelephoneContact
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Please enter the family name: ");
            string family = Console.ReadLine();
            Console.Write("Please enter the given name: ");
            string given = Console.ReadLine();
            Console.Write("Please enter the telephone number: ");
            string telephone = Console.ReadLine();

            Person demoPerson = new Person(family, given, telephone);
            Console.WriteLine("     Family Name: {0}", demoPerson.FamilyName);
            Console.WriteLine("      Given Name: {0}", demoPerson.GivenName);
            Console.WriteLine("Telephone Number: {0}", demoPerson.TelephoneNumber);
        }
    }
}
```

Task 2.4. Working with Classes II (Required)

Objective: Now that you have seen how to write and use a class, it is time to demonstrate that you can also complete this task.

You are required to write a class LibraryBook to store the basic information for a book that can be borrowed from a library: author, title, and call number. Each of these attributes is of type string. Attributes should be properly encapsulated and a constructor should be provided with parameters to initialise each of the three attributes. You are also required to write a Main method that creates a LibraryBook object, prompts the user for the required data, then displays it on the screen in an appropriate format.

2.6. Value Types and Reference Types

In Section 1.4 we examined how to declare variables and briefly discussed simple data types. In this chapter we have considered how to define new data types by writing classes. If you review the content we have studied until this time, you will notice that there is a slight difference in how we create variables of a simple type versus how we create variables from a class, as follows:

```
int accountNumber = 12345;              // from a simple type

Account account = new Account(54321);   // from a class
```

In particular, you will notice that the second variable, created from a class, requires the addition of the `new` keyword. The difference between the two variable declarations is required because of how the two data types work in the memory of a computer, in particular in the above example, the integer is a ***value type*** and the Account is a ***reference type***.

The difference between the two types becomes apparent when you examine how data is stored in these variables, as illustrated in Figure 2.11. Value types such as the `accountNumber` variable refer directly to the data they store. Reference types however, such as the `account` variable, store a memory address (a reference) to record where the object is stored in memory. When a reference type is accessed, the memory address is first retrieved and only then can the object be accessed.

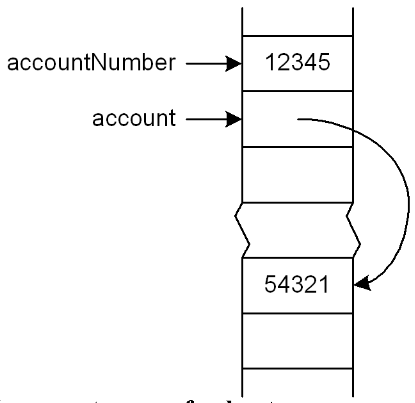


Figure 2.11: Memory storage of value types versus reference types

Interestingly, this also helps explain what happens when we don't use the combined declaration for creating objects, e.g.,

```
Account account;
account = new Account(54321);
```

Using this syntax, the first line allocates the memory slot shown immediately below the `accountNumber` variable and gives it the name `account`. The second line is divided into two parts. The code on the right of the assignment operator (=) creates the `Account` object and initialises its account number to `54321` (shown in the bottom half of the memory in the figure). The result of this object creation, using the `new` operator, is that the memory address of the new object is returned, which is then assigned to the `account` variable created in the previous step.

Identifying value types and reference types is straightforward in C# – all simple data types (see Appendix B) are value types[15]; all types defined by writing classes are reference types. The exception to this rule is the `string` data type, which although it appears to be a value type in C# is actually a reference type. This is because the `string` is considered an immutable type, i.e., a string cannot be modified after it is created. This has the effect that the `string` data type actually operates almost the same as a value type.

2.7. Enumerated Data Types

Another kind of data type that we can define is known as an ***enumerated data type***. Enumerated data types are special types which have a finite number of discrete values represented by symbolic names. The syntax for declaring a enumerated data type is as follows:

[access_modifier]`enum` <u>*enum_name*</u> `{` <u>*symbol[*</u> `=` *value][*`,` `...`*]* `}`

In the above syntax, the <u>*enum_name*</u> represents the name of the enumerated type, which follows the same rules as variables. Following the <u>*enum_name*</u> is a code block containing a comma separated list of <u>*symbol*</u> names (the symbolic/textual names for the values), which also follow the same rules as for variable names, and optionally the specification of an (integer) *value*. Two example declarations for an enumerated data type for the month of the year are as follows:

```
enum Month { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec }

enum LongMonth
{
     January = 1,
     February = 2,
     March = 3,
     April = 4,
     May = 5,
     June = 6,
     July = 7,
     August = 8,
     September = 9,
     October = 10,
     November = 11,
     December = 12
}
```

Declaring a variable of an enumerated type is the same as for simple types, i.e.,

15. User-defined structures, defined using the `struct` keyword, are also value types but are not covered in this unit.

<u>*enum_name*</u> <u>*variable_name*</u>*[* `=` *value]*`;`

How enumerated types are used in code is also the same, except for the specification of a value. Unlike other types, e.g., an integer where numbers are used, an enumerated type's value is specified as follows:

<u>*enum_name*</u>`.`<u>*symbol*</u>

To continue the month example used above, the following lines could be used to create and initialise a variable of enumerated data type:

```
Month firstExample = Month.Mar;

LongMonth secondExample = LongMonth.September;
```

2.8. Increment, Decrement, and Compound Operators

We have already seen many operators that are defined in the C# programming language (which are summarised in Appendix E). However it is worth highlighting a few operators whose functionality are used regularly in programming and save time. The first operators to consider are the increment and decrement operators. To ***increment*** a value means to add one to the value, i.e., the following line increments the variable count:

```
count = count + 1; // increment count
```

Similarly, to ***decrement*** means to subtract one from the value, as follows:

```
count = count - 1; // decrement count
```

The C# programming language provides increment (`++`) and decrement (--) operators that simplify the above two lines of code, as follows:

```
++count; // prefix increment count
count++; // suffix increment count
--count; // prefix decrement count
count--; // suffix decrement count
```

Importantly, two versions of the increment and decrement operators are shown above: as a prefix (where the operator appear immediately before the variable name) and as a suffix (after the variable name). For simple statements such as those above, there is no practical difference between the different versions. However when an increment/decrement is used in a condition statement, such as in an `if` statement, the difference is critical. If an increment/decrement is evaluated, if the prefix version is used the increment/decrement will evaluate to the new value of the variable. If the postfix version is used, the increment/decrement will evaluate to the old value of the variable, e.g.,

```
count = 5;
if(++count > 5)
     ... // true statement

count = 5;
if(count++ > 5)
     ... // true statement
```

The above two `if` statements appear to be much the same, however only the true statement in the first example will be executed. This is because `++count` will evaluate to the value `6`, the new value of `count`, whereas `count++` evaluates to the value of `5`, the previous value of `count`.

The above operators simplify incrementing and decrementing, however if you want to add/ subtract a value other than one, ***compound operators*** are used instead. The following examples summarise the compound operators provided by C#[16] (an example line of code followed by an equivalent line using a compound operator):

```
x = x + value; // add value to x
x += value;    // add value to x

x = x - value; // subtract value from x
x -= value;    // subtract value from x

x = x * value; // multiply x by value
x *= value;    // multiply x by value

x = x / value; // divide x by value
x /= value;    // divide x by value

x = x % value; // remainder after division of x by value
x %= value;    // remainder after division of x by value
```

2.9. Composite Formatting

Composite formatting, or the use of format strings, is used by many different operations in C#/ Microsoft.Net. In particular, wherever a stream of textual data can be written, including the console, files, and network communications among others, you will usually find support for composite formatting. In this section we consider composite formatting in the context of the `Console.WriteLine()` statement, however the same concepts apply anywhere composite formatting can be used in C#/Microsoft.Net.

Composite formatting begins with the introduction of a format item. Consider the following statement:

```
Console.WriteLine("Hello {0}, how are you?", name);
```

In the above statement, the `"Hello {0}, how are you?"` represents a composite format string and the `{0}` is the format item. Each format item is replaced by one of the parameters to the `WriteLine` appearing after the format string. Each parameter following the format string is numbered beginning with zero, thus `{0}` refers to the first parameter after the format string, `{1}` refers to the second parameter, and so on. Combining several of these format items together with escape sequences, it is possible to create more complex output, e.g.,

```
Console.WriteLine("{0}\t{1}({2})\t{3}, {4}", student.ID, student.Mark,
          student.Grade, student.FamilyName, student.GivenName);
```

which would produce example output similar to:

```
900123456       HD(85)  Jones, Alex
```

However, using these mechanisms we only have limited control over the output. Consider what happens if we repeat the same line twice, but change the student ID to 123456:

```
900123456       HD(85)  Jones, Alex
123456  HD(85)  Jones, Alex
```

As you can see, the output does not line up as expected. The reason for this is because a horizontal tab (`\t`) will move the cursor to the next horizontal step of 8 characters, i.e., horizontal tabs are set at the 1st, 9th, 17th, 25th, 33rd, 41st, and so on columns of the console output. On the first line, the ID 900123456 moves the cursor beyond the 9th column tab stop, so the horizontal tab moves the cursor to the 17th column. On the second line, the ID 123456 does not reach the 9th column, so the horizontal tab moves the cursor to the 9th column. The

16. There are also other compound operators, e.g., bitwise operators, that are beyond the scope of this unit.

solution to this problem is to introduce field widths, which we do by changing our `WriteLine` statement as follows:

```
Console.WriteLine("{0,10}\t{1}({2})\t{3}, {4}", student.ID, student.Mark,
         student.Grade, student.FamilyName, student.GivenName);
```

You will notice that the first format item is now indicated as `{0,10}`, where the value of 10 indicates that the minimum field width should be 10 characters. A width of 10 was selected because the largest student ID possible at Deakin was from the year 2000, where the ID number was similar to 2000123456 (10 digits long). By introducing the field width, our output now shows as follows:

```
 900123456     HD(85)  Jones, Alex
    123456     HD(85)  Jones, Alex
```

You will notice that the student ID is aligned to the right. To align the field to the left, change the field width to a negative, i.e.,

```
Console.WriteLine("{0,-10}\t{1}({2})\t{3}, {4}", student.ID,
         student.Mark, student.Grade, student.FamilyName,
         student.GivenName);
```

which results in the following output

```
900123456      HD(85)  Jones, Alex
123456         HD(85)  Jones, Alex
```

Finally, it is also possible to change how the data is displayed within a field. Consider the following two lines of code:

```
decimal price = 15.50M;
Console.WriteLine("The price is ${0}", price);
```

With the above code, we get the appearance of a currency value, as follows:

```
The price is $15.50
```

However, this is not particularly accurate way of showing a currency value. For example, consider what happens if there are more than three decimals:

```
decimal price = 15.505M;
Console.WriteLine("The price is ${0}", price);
```

Which results in:

```
The price is $15.505
```

Clearly this is not correct, the value should have been rounded to two decimal places. We could use mathematics functionality for this purpose, however this only partly solves the problem. Consider what happens if the local currency is not measured in dollars, e.g., Britain uses pounds (£), the European Union uses the Euro (€), Japan uses Yen (¥), and so on. Instead, we can use built in formatting as follows:

```
decimal price = 15.505M;
Console.WriteLine("The price is {0:c}", price);
```

You will notice that the format item has changed to '`{0:c}`'. The '`:c`' part tells the system to format the data as a currency type. By default, the formatting for a currency is determined from the regional settings in Windows[17] which are usually configured when Windows is first installed on a computer. This changes the output to:

```
The price is $15.51
```

which is what the user would expect (for a computer configured for Australia). A list of formatting codes is provided in Appendix D.

2.10. Logic Errors and Debugging

In Section 1.9 we examined syntax errors, or errors that occur in the syntax of a program that we are writing. Importantly, once a program compiles, this does not mean that that there are no more errors in the program. Errors can occur in the logic of the program that you have written, i.e., the program compiles correctly, and the program runs successfully, but there are problems with the output or behaviour of the program. Many logic errors are simple to diagnose and correct, particularly as you gain experience, however you will often come across logic errors that are harder to fix. There are a number of techniques to deal with these problems, each of which requires you to have a good understanding of the programming language and the operation of the application you are working with. The better your understanding, the quicker it will be to diagnose and correct logic errors.

Debuggers are the primary tool used for finding and diagnosing hard to find logic errors. Almost every software development platform will provide a debugger of some sort, regardless of language, compiler, IDE, and so on. The exact functionality of a debugger will vary slightly depending on the platform, but the following concepts can usually be found:

- A ***breakpoint*** is a line of code you select in a program where execution will be interrupted/paused to allow the developer to view, and in some debuggers modify, the state of the running program, i.e., to view the values of variables throughout the program;
  - Many debuggers also have the concept of a conditional breakpoint, where the code will only be interrupted/paused if a particular condition is satisfied, such as how many times the breakpoint has been reached, a variable has a particular value, etc.
- A ***watch variable*** is a variable that is constantly monitored/displayed throughout a debugging session, usually used to monitor variables that contain data critical to the correct operation of an application or variables that are receiving incorrect data in the application;
- The ***call stack*** shows the order in which methods were called to reach the current line of execution, starting with the Main function at the bottom of the call stack;
- ***Run/start*** is a function of a debugger which begins the execution of the application, which will continue until a breakpoint is reached;
- ***Continue*** is a function of a debugger which will continue execution of the application from the current line until the next breakpoint is reached;
- ***Step*** is a function of a debugger that allows you to execute a small number of lines at a time, although usually a single line of code:
  - ***Step into*** executes one line of code, and if the line of code is a method call the debugger will follow the execution "into" the method, continuing debugging from the

17. In older versions of Windows, these settings can usually be found by opening Control Panel and starting the *Regional Settings* applet. In newer versions of Windows the same settings are usually found by clicking the link *Change the date, time, or number format* link found under the *Clock, Language, and Region settings.*

first line of code inside the method;

- ***Step over*** executes one line of code, and if the line of code is a method, the debugger will complete the execution of that method in one step and continue debugging from the next line appearing after the method call;
- ***Step out*** will complete the execution of the current method in one step and continue debugging from the first line of code after the original invocation of the method;

Task 2.5. Logic Errors (Required)

Objective: Logic errors are bugs in a program that you will need to become very efficient at correcting. This task will give you some experience wiht the debugger for diagnosing and correcting logic errors.

In this exercise, you are required to remove the logic errors from the provided program. The program is designed to record the deposits and withdrawals for a simple bank account, calculating interest on a daily basis. At the end of the month (assumed to be the 31st day), the interest is automatically added to the deposits for that day. Importantly, the deposits for each day are processed before any withdrawals. Thus, with a zero balance, a deposit of $10.00 and withdrawal of $10.00 is able to be processed. If, after processing the deposits, there is not enough money to process the withdrawals, all withdrawals are rejected and a zero total recorded. The correct output for this program is as follows:

```
+-----+-------------+-------------+-------------+
| Day |    Deposits |  Withdrawals |     Balance |
+-----+-------------+-------------+-------------+
|   1 |       $1.00 |       $0.10 |       $0.90 |
|   2 |       $2.00 |       $0.20 |       $2.70 |
|   3 |       $3.00 |       $0.30 |       $5.40 |
|   4 |       $4.00 |       $0.40 |       $9.00 |
|   5 |       $5.00 |       $0.50 |      $13.50 |
|   6 |       $6.00 |       $0.60 |      $18.90 |
|   7 |       $7.00 |       $0.70 |      $25.20 |
|   8 |       $8.00 |       $0.80 |      $32.40 |
|   9 |       $9.00 |       $0.90 |      $40.50 |
|  10 |      $10.00 |       $1.00 |      $49.50 |
|  11 |      $11.00 |       $1.10 |      $59.40 |
|  12 |      $12.00 |       $1.20 |      $70.20 |
|  13 |      $13.00 |       $1.30 |      $81.90 |
|  14 |      $14.00 |       $1.40 |      $94.50 |
|  15 |      $15.00 |       $1.50 |     $108.00 |
|  16 |      $16.00 |       $1.60 |     $122.40 |
|  17 |      $17.00 |       $1.70 |     $137.70 |
|  18 |      $18.00 |       $1.80 |     $153.90 |
|  19 |      $19.00 |       $1.90 |     $171.00 |
|  20 |      $20.00 |       $2.00 |     $189.00 |
|  21 |      $21.00 |       $2.10 |     $207.90 |
|  22 |      $22.00 |       $2.20 |     $227.70 |
|  23 |      $23.00 |       $2.30 |     $248.40 |
|  24 |      $24.00 |       $2.40 |     $270.00 |
|  25 |      $25.00 |       $2.50 |     $292.50 |
|  26 |      $26.00 |       $2.60 |     $315.90 |
|  27 |      $27.00 |       $2.70 |     $340.20 |
|  28 |      $28.00 |       $2.80 |     $365.40 |
|  29 |      $29.00 |       $2.90 |     $391.50 |
|  30 |      $30.00 |       $3.00 |     $418.50 |
|  31 |      $31.67 |       $3.10 |     $447.07 |
+-----+-------------+-------------+-------------+
|     |     $496.67 |      $49.60 |             |
+-----+-------------+-------------+-------------+
```

Two code files are provided for this exercise: BankAccount.cs and MonthlyStatement.cs. These program files have been provided in DSO, separate to the workbook, to save space. The BankAccount.cs file, containing the Main method, does not contain any logic errors. The MonthlyStatement.cs file contains a single class MonthlyStatement, which does contain logic errors.

Note that the provided program deliberately contains concepts that we have not yet covered in this unit, in particular arrays and the `StringBuilder` class. This is a situation you will regularly find yourself in industry, and it is vital that you learn how to work with code you don't fully understand. Although you could read the textbook or the web to learn about arrays and the `StringBuilder`, try to avoid this if possible as you will not always have the time to do so.

Task 2.6. Term Definition (Required)

Note: The submission requirements for this task are only that you complete definitions for the underlined terms. However note that you are required to be familiar with the remaining terms.

From your own research and the information above, define the following terms in the glossary (Appendix A): access modifier, accessor method, algorithms, argument list, assembly, breakpoint (debugger concept), call stack (debugger concept), composite formatting, compound operator, condition, conditional AND, conditional OR, continue (debugger concept), dangling else, decrement, enumerated data type, equality operator, increment, instance method, instance variable, iteration, logical negation, logical operator, loop body, mutator method, property, reference type, relational operator, repetition control structure, return value, run/start (debugger concept), selection control structure, short-circuit evaluation, step into (debugger concept), step out (debugger concept), step over (debugger concept), structure theorem, truth tables, value type, and watch variable (debugger concept).

2.11. Further Problems from the Textbook

The textbook includes many problems useful for practicing the concepts examined in this session. Below is a list of recommended problems you should try from the textbook for further study and practice. Even if you don't complete these exercises, you should still review/read them to ensure you at least have an idea of the answer for each problem. If not, you have not studied the materials adequately and should continue your research. The recommended problems for this session are as follows:

- pp121, Exercise 3.17.
- pp168, Exercises 4.10-14.
- pp214-9, Exercises 5.15-23, and 5.25-33 (for Exercise 5.17 you may find it easier to use kilometres instead of miles and litres instead of gallons).
- pp266-9, Exercises 6.9-19, and 6.22-27.

2.12. Past Exam Questions

Semester 2, 2004 - Question 1

Define the following TEN (10) terms. Be careful not to use the terms in your definitions.

(f) Reference
(h) Constructor

Semester 2, 2005 - Question 1

Define the following terms:

(d) Accessor
(b) Encapsulation

Semester 2, 2007 - Question B1

(d) Define the debugging concepts of breakpoints, watch variables, and step over.

Semester 2, 2007 - Question B2

(d) Discuss the need for accessors and mutators. Discuss properties as an alternative to accessors and mutators.

Semester 2, 2008 - Question 1[18]

(c) Define the three levels of accessibility in C#: public, ~~protected~~, and private.

Trimester 2, 2009 - Question 1

Define briefly the following terms:

(f) Structure theorem
(j) Short-circuit evaluation

Trimester 2, 2009 - Question 2

(a) An object-oriented application consists of several interacting objects. Is this an accurate statement? Explain.
(b) Briefly discuss the following two statements:
  (i) A class is a template for an object; and
  (ii) ~~An abstract class is a template for a concrete class.~~

Trimester 2, 2009 - Question 4

For this task you are required to write the code for a C# program for an airline's passenger management software, consisting of the following classes:

- ...
- Ticket class – provides the information about an individual ticket including seat assignment, class, ~~and the passenger if the ticket is booked~~;
- Person class – providing the information about an individual, including name and contact number; and
- ...

Note: Large parts of this question have been omitted as it draws on content that has not yet been covered. However you should be able to see that the concepts covered so far are fundamental to developing object-oriented applications.

18. Note that 'accessibility levels' is another way to refer to access modifiers, presented in Section 2.3. In previous years only the private, protected, and public access modifiers were considered.

Trimester 2, 2010 - Question 1

Briefly define the following terms:

(b) Structure theorem

Trimester 1, 2011 - Question 1

Briefly define the following terms:

(a) Short-circuit evaluation

Trimester 1, 2011 - Question 2

(b) Describe how the members you define in a class are used to construct the interface to instances of that class.