

Session 3. Methods and arrays

In this session we complete our review of the pre-requisite content covering arrays, a more detailed review of methods, and scope among other topics. We also examine some of the more obscure/advanced topics, which may or may not have been covered in pre-requisite studies, but are common in object-oriented programming such as method overloading.

Session Objectives

In this session, you will learn:

- | The concept of a collection and how to use them in your programs;
- | Static members are attributes and operations that can be used in the absence of an object;
- | More about methods: using parameters for both input and output, how defining methods with the same name, and recursive methods as an alternative to iteration;
- | The concept of scope and how it is applied in the C# programming language; and
- | How to validate input that is entered by the user.

Unit Learning Outcomes

- | *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*

We will not be directly addressing this ULO, however we continue working with object-oriented programs which reinforce understanding of these concepts.

- | *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*

We continue reviewing the C# programming language in this session.

- | *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*

Although we do not directly address this ULO, knowledge of the C# programming language is critical to this outcome.

- | *ULO 4. Construct object-oriented designs and express them using standard UML notation.*

This ULO is outside the scope of this session.

Required Reading

Additional reading is required from the textbook Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014, as follows:

Required Reading: Chapter 7

This chapter takes a closer look at the finer details of defining methods, including parameter passing and types of passing, how the method call stack

works, method overloading, and recursive methods. The chapter also considers a number of important topics outside of methods, including static members, argument promotion and casting, scope, and the Microsoft.Net framework support for random number generation and mathematics.

Required Reading: Chapter 8

This chapter introduces arrays, a fundamental collection provided by most programming languages. The foreach repetition structure is also introduced in this chapter, and finally finishes with an examination of variable-length argument lists and command line arguments, both of which require the use of arrays. Read the following sections carefully: Sections 8.1-8.11 and Section 8.13.

Required Reading: Chapter 9

These chapters is primarily focussed on Linq, which is beyond the scope of this unit. However the chapter also introduces generic collections, which we do examine. Read Section 9.4 carefully.

Relevant Web Resources

The following online videos may provide additional insight to this week's content:

- ▮ **[Video]** C# 4.0 New Features - Named Parameters
<https://channel9.msdn.com/Blogs/mike+ormond/C-40-New-Features-Named-Parameters>
- ▮ **[Video]** C# 4.0 New Features - Optional Parameters
<https://channel9.msdn.com/Blogs/mike+ormond/C-40-New-Features-Optional-Parameters>

Required Tasks

The following tasks are mandatory for this week:

- ▮ Task 3.1. Recursive Methods
- ▮ Task 3.2. Standard Deviation
- ▮ Task 3.3. Validity Checking

3.1. Introduction

In previous sessions we introduced the object-oriented development methodology and the C# programming language. In this session, we develop our knowledge further by learning about collections of data and more about methods. We begin our study by examining the concept of a collection, in particular the array and (Microsoft.Net) generic collections, before examining the foreach repetition control structure for processing data stored in collections and command line arguments as an example. Static and constant members are then examined, which allow attributes and operations to be defined that are part of a class, rather than part of an object, and as such can be used in the absence of an object and/or global to all objects of a particular

class. This is followed by some of the finer details of defining methods, in particular passing parameters by-value and by-reference (allows data to be input and output from methods), method overloading (allows several methods with the same name to be defined in a single class), and recursive methods (allows repetition without using iteration). We then consider scope, a concept in software development that is critical to understand as it defines the limits of a variable name, method name, and so on, i.e., it defines where that name can be used in your code. Data type conversion is revisited to consider argument promotion (implicit data type conversion) and casting (explicit). Casting is a more efficient way of converting data than the Convert utility, however it has limitations. We complete our study in this session with an examination of the Microsoft.Net framework support for Mathematics functionality and the basics of input validation.

3.2. Collections

In Section 1.4 we considered how to create and use variables for storing a single element of data. For many applications there is also a need to be able to store several elements of data, for which we use a collection. We examine two types of collections in this unit: the array and the generic collection. Arrays are found in almost every programming language and provide a fixed length collection of elements¹. Generic collections are provided by the Microsoft.Net framework² and provide a variable length collection of elements. Finally, we consider the `foreach` loop, a repetition control structure that simplifies reading/processing of data in collections and command-line arguments as an example application of arrays.

3.2.1. Arrays

Arrays are simple collections of data which are of fixed length. There are three types of arrays you will encounter:

- | A one-dimensional array (Figure 3.1a) is a simple array that contains a number of elements, e.g., an array containing 5 elements;
- | A multi-dimensional array³ (Figure 3.1b) is an array that contains elements in more than one dimension, e.g., a two-dimensional array with 5 rows and 4 columns (20 elements in total);
- | A jagged array (Figure 3.1c) is similar to a multi-dimensional array, except that the number of elements in a particular dimension do not need to be equal, e.g., a two-dimensional jagged array with 3 rows with 6, 4, and 5 columns (15 elements in total).

¹ Some software development platforms provide variable length arrays, however these are rare.

² Similar functionality is often found in other software development platforms, e.g., the Java Collections Framework (Java), the Standard Template Library (C++), and so on.

³ Multi-dimensional arrays are sometimes referred to as n-dimensional arrays.

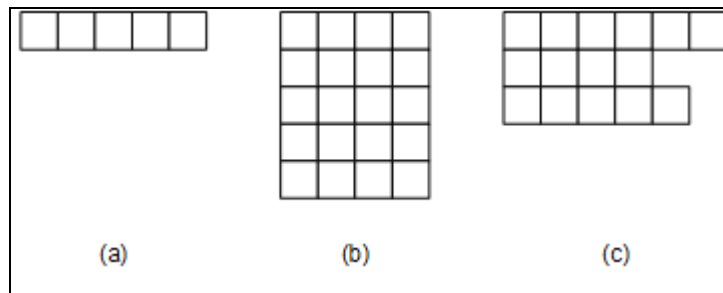


Figure 3.1: Visual Structure of Array Types

The syntax for declaring a one-dimensional array is as follows:

```
type [] name = new type[size];
```

or separating the declaration and creation of the array:

```
type [] name;  
name = new type[size];
```

There are two parts to the above statement, the declaration of an array variable (to the left of the equals symbol in the first syntax, or the first line in the second syntax) and the creation of the array (to the right of the equals symbol in the first syntax, or the second line in the second syntax). In the above syntax, `type` represents the data type of the elements to be stored in the array and appears both in the variable declaration and the array creation. The square brackets (`[]`) in the declaration indicate that this declaration is for an array, not a single variable, and the square brackets in the array creation contain the size of the array to be created. The `name` is the name of the variable and must follow the same rules (see Section 1.4). An example matching Figure 3.1a is as follows:

```
int [] simpleArray = new int[5];
```

The syntax for declaring a multi-dimensional array is as follows:

```
type [, [, [...]]] name = new type[size1, size2[, ...]];
```

or

```
type [, [, [...]]] name;  
name = new type[size1, size2[, ...]];
```

The syntax for a multi-dimensional array is almost identical to a one-dimensional array, except the contents inside the square brackets changes. In the variable declaration, the brackets contain a comma (,) to indicate the second and further dimensions of the array, i.e., for a two-dimensional array the declaration will have `[,]`, for a three-dimensional array the declaration will have `[, [,]]`, and so on. The square brackets in the array creation represents a comma separated list of the sizes for each dimension of the array. An example matching Figure 3.1b is as follows:

```
int [, ] multiArray = new int[5,4];
```

Finally, the syntax for declaring a jagged array⁴ is as follows:

```
type [][] name = new type[rows][];
name[0] = new type[size1]; name[1]
= new type[size2];
[...]
```

or

```
type [][] name;
name = new type[rows][];
name[0] = new type[size1];
name[1] = new type[size2];
[...]
```

The syntax for declaring a jagged arrays is quite different to one-dimensional and multi-dimensional arrays. Note the use of several sets of square brackets, both in the variable declaration and array creation part. The first line of the above syntax creates the rows of the jagged array, the number of rows being specified by *rows*. The second and further statements create the elements for the individual rows, which do not need to be equal in length. An example matching Figure 3.1c is as follows:

```
int [][] jaggedArray = new int[3][];
jaggedArray[0] = new int[6];
jaggedArray[1] = new int[4];
jaggedArray[1] = new int[5];
```

Like all objects in C#, a number of useful methods and properties are provided by array objects⁵. The following list includes the more commonly used members:

┆ Properties:

┆ Length—the total number of elements in the array (in all rows, columns, etc.);

┆ Rank—how many dimensions the array has, i.e., a one-dimensional array will have a rank of 1, a two-dimensional array a rank of 2, and so on.

┆ Methods:

┆ GetLength(*dimension*)—get the length for a specific *dimension*, e.g., how many elements in a row;

Using these array members, we can use for loops to access the elements stored in an array, as follows for a one dimensional array:

```
for (int i = 0 ; i <= simpleArray.GetLength(0) ; i++)
    statement; // individual element is simpleArray[i]
```

Or for a multidimensional array:

```
for (int i = 0 ; i <= multiArray.GetLength(0) ; i++)
    for (int j = 0 ; j <= multiArray.GetLength(1) ; j++)
        statement; // individual element is multiArray[i,j]
```

⁴ Note that we only consider two-dimensional jagged arrays in this unit.

⁵ Full documentation of array methods and properties can be found at [https://msdn.microsoft.com/en-us/library/system.array_members\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/system.array_members(v=vs.90).aspx)

And finally for a jagged array:

```
for (int i = 0; i < jaggedArray.GetLength(0); i++)
    for (int j = 0; j < jaggedArray[i].GetLength(0); j++)
        statement; // individual element is jaggedArray[i][j]
```

An example program that uses arrays is shown in Figure 3.2. This example has been adapted from the multiplication table example shown previously in Figure 2.4. This new version allows the user to enter the size of the multiplication table that they wish to generate. A two-dimensional array is then created, and the values are stored into the array using two for loops, one loop for the rows of the table and the second loop, nested in the first for loop, to generate the result for each column in a row. Once the multiplication is generated, the program then uses a similar set of two for loops to display the multiplication table, which is almost the same loop as was shown previously in Figure 2.4 except the data is retrieved from the two-dimensional array rather than being calculated. Note the use of the `GetLength()` method of the arrays in the example to determine the size of a particular dimension.

```
/* *****
** File: ArrayTimesTable.cs
** Author/s: Justin Rough
** Description:
**     A more advanced version of the TimesTable.cs program
** which used loops to produce a multiplication table. This
** new version uses an array to contain the times table, the
** size for which is obtained from the user.
** ***** */
using System;

namespace ArrayTimesTable
{
    class ArrayTimesTable
    {
        static void Main(string[] args)
        {
            // Determine what size times table the user wants
            Console.WriteLine("How many rows? ");
            int rows = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("How many columns? ");
            int columns = Convert.ToInt32(Console.ReadLine());

            // Build the array
            int[,] timesTable = new int[rows, columns];
            for (int i = 0; i < timesTable.GetLength(0); i++)
                for (int j = 0; j < timesTable.GetLength(1); j++)
                    timesTable[i, j] = (i + 1) * (j + 1);

            // Display the array contents
            for (int i = 0; i < timesTable.GetLength(0); i++)
            {
                for (int j = 0; j < timesTable.GetLength(1); j++)
                {
                    Console.Write("{0,3} ", timesTable[i, j]);
                }
                Console.WriteLine();
            }
        }
    }
}
```

```

    }
}

```

Figure 3.2: Multiplication Table using Arrays (ArrayTimesTable.cs)

When working with arrays it is important to remember that one of the key advantages of the array is that it allows elements to be accessed directly, e.g., you can access the first element (`array[0]`), tenth element (`array[9]`), or thousandth element (`array[999]`) without needing to do any additional processing. This is not the case with many other data structures, including many of the generic collections often provided by languages (addressed in Section 3.2.2)⁶. For this functionality to be provided, it is necessary for the array to be stored contiguously in memory, i.e., in one single chunk of memory. The problem with contiguous storage however, is that it is very difficult/impossible to change the size of the array later as the memory next to the array will likely be taken by another object. For this reason most programming languages, including C#, specify that arrays are fixed size.

We can overcome the fixed size by allocating a new array and copying any data to the new array, as illustrated in Figure 3.3. The original array, shown on the left, has run out of capacity to store any elements. A new array with additional slots, shown on the right, is allocated to replace the original array. All of the elements are copied from the old array to the new array, and the old array is no longer used. Programming languages will often provide utility methods to help the programmer do this⁷.

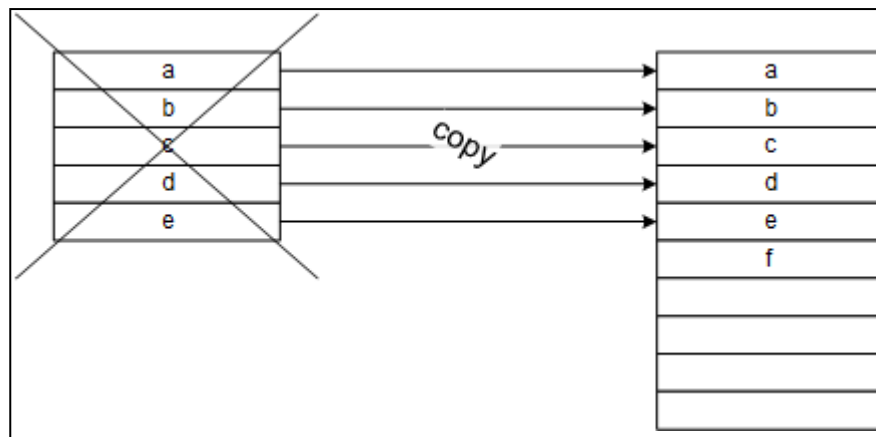


Figure 3.3: Overcoming the Fixed Array Size Limitation

3.2.2. Generic Collections

We have seen in the previous section how arrays suffer the problem of having a fixed size and how to overcome this problem. Although this functionality is not particularly complicated, it is one of a number of data structures and algorithms that are needed regularly by programmers. Rather than having to redevelop this functionality every time, the Microsoft.Net framework provides a ready made class for this purpose,

⁶ The study of different data structures, their advantages and disadvantages, is the topic of the unit SIT221 Classes, Libraries and Algorithms.

⁷ Microsoft.Net provides the `Array.Resize()` method for this purpose, however we do not examine it in this unit.

known as the generic `List` class. We examine generics in detail in Section 11.3, however simply put, generics allow classes and methods to be defined that operate on any data type. To use the `List` class, the first step is to indicate the required namespace at the top of the C# program file⁸:

```
using System.Collections.Generic;
```

The `List` is then declared using the following syntax is used:

```
List<type> name = new List<type>();
```

For example, to create a `List` for storing integers called `numberList`, the following declaration is used:

```
List<int> numberList = new List<int>();
```

Note that it is not possible to create multi-dimensional lists or jagged lists, as we saw for arrays, however there are alternatives, e.g., it is possible to create a "`List` of `List` objects" to provide equivalent functionality. The `List` class defines a number of useful properties and methods, as follows:

┆ **Properties:**

- ┆ `Count` – the total number of elements in the list;

┆ **Methods:**

- ┆ `Add(element)` – store an element at the end of the collection;
- ┆ `Clear()` – remove all elements in the collection/reset the collection;
- ┆ `Contains(element)` – searches the collection for the element and returns true if it's found (false otherwise);
- ┆ `Insert(index, element)` – inserts an element into the list at the indicated index;
- ┆ `Remove(element)` – searches the collection for the element and removes it from the list if found, returning true if the element was found and removed (false otherwise);
- ┆ `RemoveAt(index)` – removes the element from the collection that is stored at the specified index;
- ┆ `Sort()` – sorts the elements stored in the list in ascending order;

It is also possible to retrieve elements from the `List` by using the index operator, just like for arrays, e.g., to obtain the fifth element from the `numberList` declared above, you would use `numberList[4]`. An example of using a `List` is shown in Figure 3.4, described in Section 3.2.3, below.

Note that the Microsoft.Net framework includes a number of other collections, however we do not examine them in this unit. Other types of collections and their applications are considered in the unit SIT221 Classes, Libraries and Algorithms.

⁸ The `System.Collections.Generic` namespace is added by the Visual Studio environment by default, as the `List` and other generic collections are used regularly in programming.

3.2.3. The foreach Loop

The `foreach` loop is another repetition control structure that is used to access the elements stored in a collection one at a time. The syntax for a `foreach` loop is as follows:

```
foreach (type name in collection_name)  
    statement;
```

or using a code block if multiple statements are required:

```
foreach (type name in collection_name)  
{  
    statement1;  
    statement2;  
    ...  
}
```

In the above syntax, *type* represents the data type stored in the collection, e.g., for an array of integers the type would be `int`. The *collection_name* is the name of the collection variable, i.e., the name of the array, `List`, or other generic collection. The *name* variable is then what is used inside the `foreach` loop to access each individual element stored in the collection in turn.

Critically, no modifications should be made to the collection that is being processed by a `foreach` loop, i.e., you should not add or remove elements to/from the same collection being considered by the `foreach` loop. This is because the behaviour of the `foreach` loop is undefined when the collection is modified⁹, i.e., modifying a collection is not expected and can lead to errors in your application that result in unusual behaviour and difficult to diagnose. Note however that this rule does not prevent modification of a different/separate collection from a `foreach` loop, e.g., a `foreach` loop could be used to copy all elements from one collection to another matching a particular criteria such as:

```
foreach (Student s in StudentList)  
{  
    if(s.Result >= 50)  
        PassedStudentList.Add(s);  
    else  
        FailedStudentList.Add(s);  
}
```

An example of using the `foreach` loop and the `List` generic collection is shown in Figure 3.4. In this example, the program prompts the user for a string and then stores the retrieved string in a `List` of strings. This continues until the user types in the value of `end` (upper or lower case), at which point the program will sort the `List` of strings and output the list of sorted strings results to the terminal.



⁹ Most collections will actually throw an exception (`InvalidOperationException`) if the collection is modified during a `foreach` loop.

```
** File: StringSorter.cs
** Author/s: Justin Rough
** Description:
**     A simple program that reads strings entered by the user
** and stores them in a List of strings until the user enters
** a value of END. The program then sorts the list of values
** and outputs the result.
*****/
using System;
using System.Collections.Generic;

namespace StringSorter
{
    class StringSorter
    {
        static void Main(string[] args)
        {
            List<string> stringList = new List<string>();
            string input;

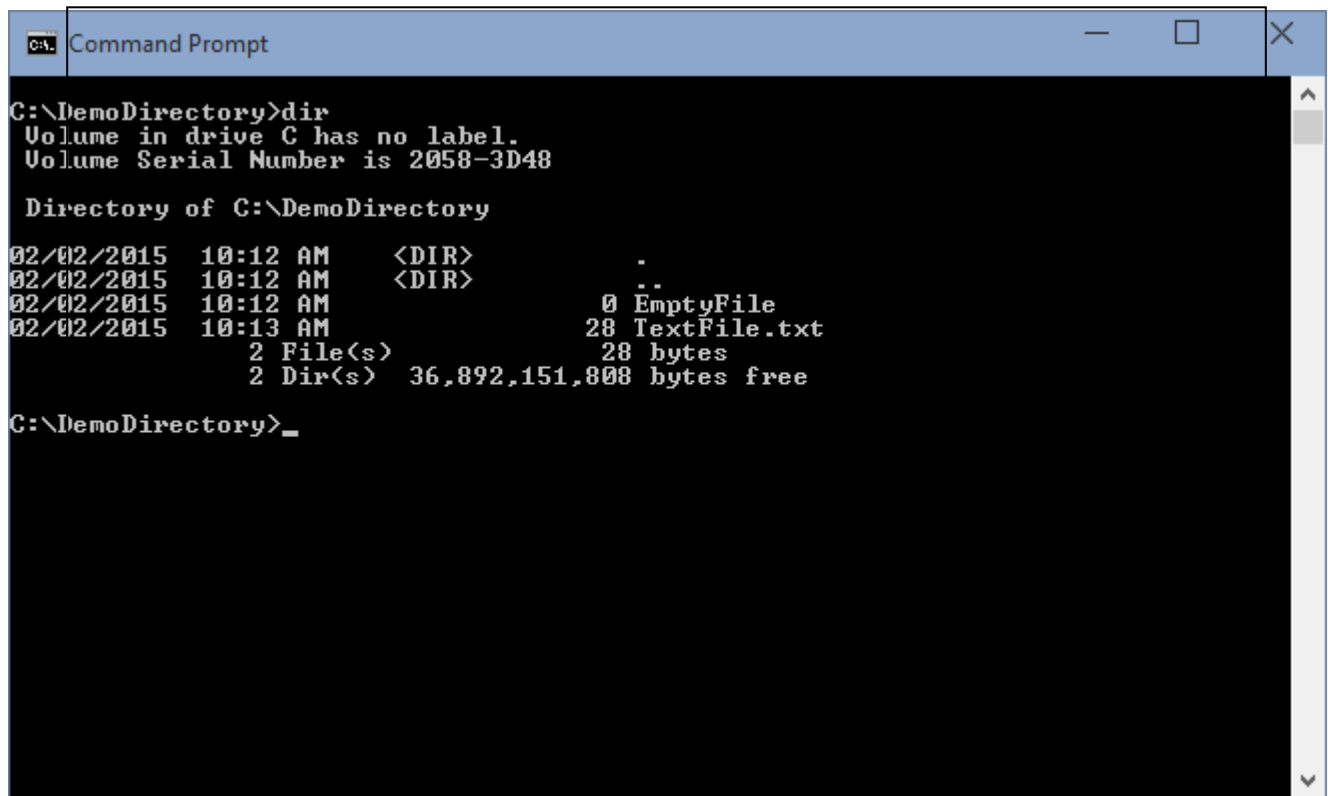
            Console.Write("Please enter a string or END to finish: ");
            input = Console.ReadLine();
            while (input.ToUpper() != "END")
            {
                stringList.Add(input);
                Console.Write("Please enter a string or END to finish: ");
                input = Console.ReadLine();
            }

            stringList.Sort();
            Console.WriteLine("Sorted strings:");
            foreach (string s in stringList)
                Console.WriteLine("\t" + s);
        }
    }
}
```

Figure 3.4: Example of the List Class and foreach Loop (StringSorter.cs)

3.2.4. Command Line Arguments

In addition to receiving input from the terminal it is also possible for a program to receive input from the command line. In this unit we are writing programs that run in a console window which can be manually started, outside of Visual Studio, by clicking on the **Start** button, then **All apps**, **Windows System**, and finally selecting the **Command Prompt** menu option. In this window, we can issue a number of commands, such as the `dir` command to obtain a list of files contained in the directory, as shown in Figure 3.5.



```
C:\DemoDirectory>dir
Volume in drive C has no label.
Volume Serial Number is 2058-3D48

Directory of C:\DemoDirectory

02/02/2015  10:12 AM    <DIR>          .
02/02/2015  10:12 AM    <DIR>          ..
02/02/2015  10:12 AM                0 EmptyFile
02/02/2015  10:13 AM               28 TextFile.txt
                2 File(s)                28 bytes
                2 Dir(s)  36,892,151,808 bytes free

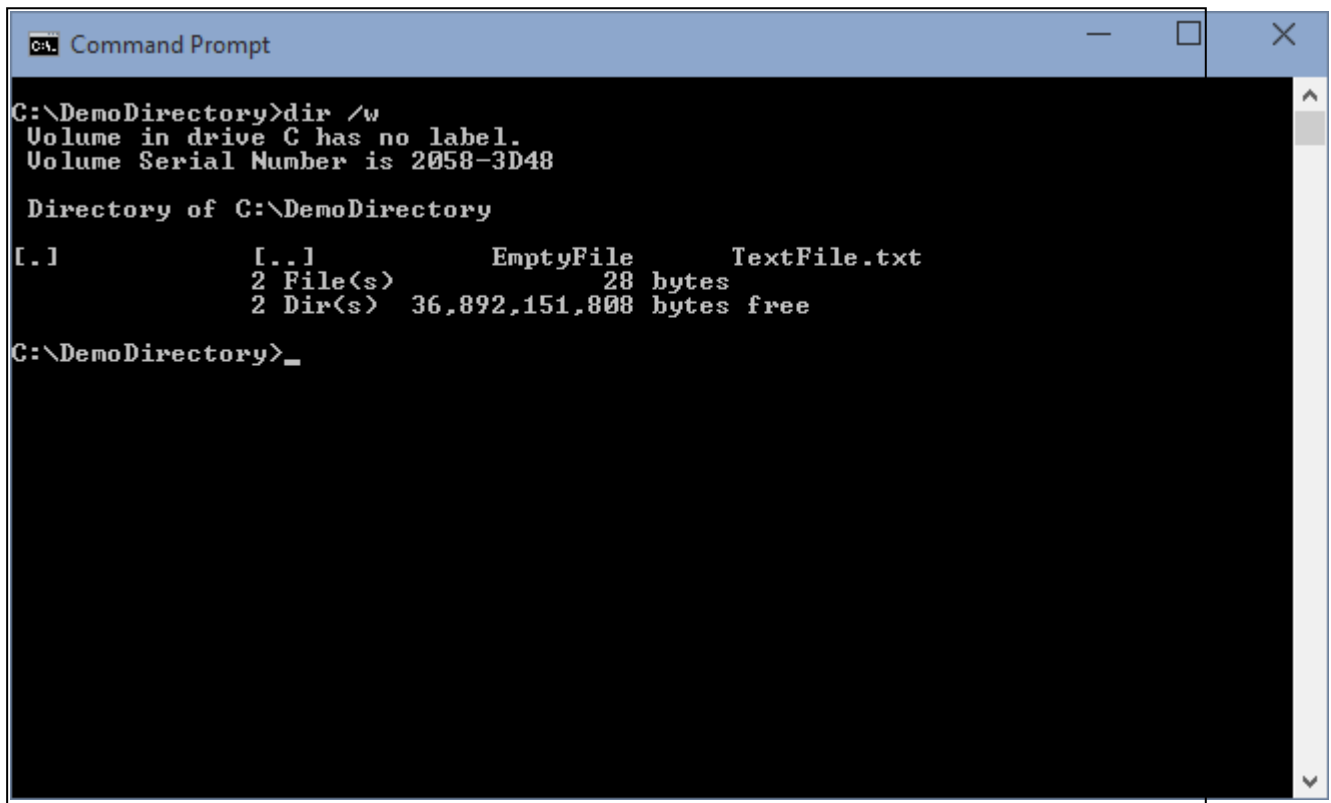
C:\DemoDirectory>_
```

Figure 3.5: Example Directory Listing in the Command Prompt

The `dir` command produces a list of files, one per line, indicating the name of each file, the date that it was created, and the size of the file. Although this program does not prompt for user input, it nonetheless can receive input through command line arguments, including

- | `/B`—bare format, i.e., only display the file name;
- | `/P`—display directory information one page at a time, pausing at the end of each page;
- | `/S`—list files in both the current directory and any subdirectories; and
- | `/W`—display the directory listing using a wide format.

Figure 3.6 shows the effect on the `dir` command when the `/w` flag is added.



```
Command Prompt

C:\DemoDirectory>dir /w
Volume in drive C has no label.
Volume Serial Number is 2058-3D48

Directory of C:\DemoDirectory

[.]                [..]                EmptyFile        TextFile.txt
2 File(s)          28 bytes
2 Dir(s)  36,892,151,808 bytes free

C:\DemoDirectory>_
```

Figure 3.6: Example Directory Listing with the /w Flag Added

You can also accept command line arguments in your own programs. Command line arguments are passed to a program as a parameter to the Main method. Reconsider the signature of the Main method:

```
static void Main(string[] args)
```

You will notice that the Main method has a single parameter, `args`, which is an array of `string` values. This array contains all of the command line arguments, and can be processed the same as any other array. As suggested by the `Main` method's signature, all command line arguments are received as `string` values, the same as other input received via the `Console` class.

Although we can clearly see how to receive command line arguments in our programs, it is not immediately apparent how to provide them using Visual Studio. To specify command line parameters in Visual Studio, click on the **Project** menu then select the **Properties** menu option, which is usually the last option in the menu. Note that the menu option will include the project name, so if your project is called `MyProject` the menu item will be shown as "MyProject Properties...". Once the project properties are shown, you will see a list of tabs down the left hand side of the window. Select the **Debug** tab and you will find a textbox where you can enter command line arguments (see Figure 3.7 for an example of the menu option and properties window). These command line arguments will always be used when running the program unless you return to the project properties window and remove them.

3.3. Static and Constant Members

We have already seen how to define operations and attributes that are part of/as-associated with an instance/object (Section 2.4 and Section 2.5). It is also possible to define operations and attributes as part of/associated with a class¹⁰, i.e., operations and attributes that are global to all objects of that class and can be used even when no objects have been created from that class. These are implemented in C# by adding the keyword `static` to the definition of instance methods, instance variables, and properties, and are usually referred to as static methods, static variables, and static properties, respectively, or collectively as static members. The same basic rules of encapsulation are applied as for instance methods, variables, and properties, i.e., static variables should be defined with a `private` access modifier and an interface defined using `public` accessor and mutator methods or public properties.

The final type of member that is part of a class is the constant, used to store values that will never change, e.g., the value of pi (π). Constants are declared the same as an instance variable, except the keyword `const` is added, their value must be provided, and it is standard practice to name constants using only capital letters (several words are separated by underscores), e.g.,

```
private const int MONTHS_PER_YEAR = 12;
```

An example using static members and constants is shown in Figure 3.7. This example shows how the interest rate for a `Mortgage` account can be stored in static members – the same interest rate applies to all `Mortgage` objects, and any change in the interest rate would be reflected immediately in all `Mortgage` objects. For this purpose, a private static variable is used to store the current interest rate, which is initialised to a default rate of 8.5% per annum, which itself is stored in a constant. A static property then provides the public interface for querying/modifying the interest rate (accessor and mutator methods could have equally been used).

```
class Mortgage
{
    ...
    private const double DEFAULT_INTEREST_RATE = 8.5; // 8.5% per annum
    private static double _InterestRate = DEFAULT_INTEREST_RATE;
    public static double InterestRate
    {
        get { return _InterestRate; }
        set { _InterestRate = value; }
    }
    ...
}
```

Figure 3.7: Example of static members used to store a mortgage interest rate

As can be seen in Figure 3.7, static members can be accessed in the same way as instance members. Outside of the class however, the class name is used instead, i.e.,

¹⁰ Note that some object-oriented textbooks may refer to static operations/methods and static attributes/variables as class operations/methods and class attributes/variables, respectively.

class_name.member_name

For example, to display the current interest rate, the following statement could be used:

```
Console.WriteLine("Interest rate is {0:p}", Mortgage.InterestRate);
```

Now that we have learned about static members, it should also now be clear why the `Main` method is declared as static, i.e.,

```
static void Main(string[] args)
```

The `Main` method is the entry-point for the application, i.e., the execution of the program begins with the `Main` method. When a program first begins execution, no objects have yet been created, thus the `Main` method is declared as static so that it can be executed without requiring an object reference.

3.4. More on Methods

In this section we examine five important areas of method development: parameter passing, method overloading, optional parameters, named parameters, and recursion. There are two alternative ways of passing data into a method: by-value, which allows a copy of data to be passed into a method, and pass-by-reference, which effectively passes the actual variable holding data to the method allowing it to be modified (hence data can potentially be passed out as well as passing data in). Method overloading allows us to define several methods with the same name, a concept that is exploited regularly in object-oriented programming. Finally, recursion refers to defining methods which include method calls to themselves, allowing repetition to be achieved without using an iteration control structure, i.e., `while`, `do-while`, `for`, or `foreach`.

3.4.1. Pass-by-Value and Pass-by-Reference

Recall from Section 2.4 that the definition of a method includes the specification of a parameter list:

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

The parameter list is a comma separated list of zero or more variables used to pass information in and out of a method. These variables can be either a simple type or defined by a class. Until now, we have only seen methods that receive information from the parameters, i.e., the parameters are inputs to the method. However, parameters can do more than provide simple input.

Almost all programming languages will provide you with two ways to pass data to a method, i.e., to specify/provide the parameters, known as pass-by-value and pass-by-

reference¹¹. Pass-by-value literally refers to passing a value to the method, the value either being provided as a literal or being copied from a variable. By passing a parameter by value, any changes made to that parameter by the method will not be visible outside of that method. This is the case even when a variable is passed, as the data is copied when using pass-by-value (hence only the copy can be changed by the method, not the original data).

Alternatively pass-by-reference passes the actual variable as the parameter instead of just the value. Note that by passing the actual variable, there is no possibility of specifying a literal for a parameter – you must store it in a variable first, then pass the variable. However pass-by-reference provides two important advantages: (i) the value does not need to be copied (required for pass-by-value), and (ii) the method can output information by changing the value stored in a variable.

Earlier in Section 2.4, we identified that parameters are specified in the argument list the same as for variable declarations except they are separated by commas and do not finish with a semi-colon. However there is also an optional keyword that can be specified first, resulting in the following syntax for specifying a *parameter*:

`[ref|out]type name`

Using this syntax we identify three different types of parameter passing in C#: input parameters, reference parameters, and output parameters. Input parameters provide pass-by-value, as presented above, and can only provide input to a method. Both literals and variables can be used for the parameter, but when a variable is used the compiler will check to make sure the variable has been initialised first (a value has been assigned to the variable prior to the method call).

Reference parameters¹² implement pass-by-reference, as presented above, provide input to a method and can optionally be used for output from the method. As for input parameters, the compiler will check to make sure that the variable has been initialised first. To specify a reference parameter, the keyword `ref` must precede the parameter in the method definition, and must also precede the variable being passed in the method invocation as well, i.e.,

```
void ExampleMethod(ref int value)           // method definition
{
    ...
}

object.ExampleMethod(ref aVariable);        // method invocation
```

Output parameters are a special type of parameter passing often found in modern programming languages. Output parameters are roughly equivalent to reference parameters, except they cannot provide input to the method, and the compiler will generate an error if the method attempts to read the value of an output parameter

¹¹ You will also see the terms call-by-value and call-by-reference used as well, which are equivalent.

¹² Note that reference parameters discussed here are separate to reference types, which will be discussed shortly.

before a value has been assigned, i.e., once the method assigns a value to an output parameter it can read its value back, but not before. Unlike input and reference parameters, the compiler does not require a variable passed as an output parameter to be initialised first. To specify an output parameter, the keyword `out` must precede the parameter in the method definition, and must also precede the variable being passed in the method invocation as well, i.e.,

```
void ExampleMethod(out int value)           // method definition
{
    ...
}

object.ExampleMethod(out aVariable);        // method invocation
```

When considering whether how to pass parameters, it is also important to consider whether value type or reference type are being passed. Value types are straightforward and operate precisely according to the definitions of input, reference, and output parameters as indicated above. However reference types are slightly more complex.

The behaviour of pass-by-value and pass-by-reference does not change for reference types as such, however recall from Section 2.6 that reference types do not store data directly, but instead store the memory address of the object. It is critical to recognise that it is in fact the memory address that is being passed-by-value or passed-by-reference, not the object itself. Thus any changes made to the object are permanent as they are not made to a copy. If you need to use a copy of an object, you must manually copy the object first. We examine how to copy objects in Section 4.4.1. Passing a reference type using pass-by-reference means that the method can replace the object being referred to by the parameter if necessary.

3.4.2. Method Overloading

In examining methods we have so far examined the syntax for defining a method, how data is provided as input and output for a method, and the different types of parameters that are used in defining the method. In Section 2.4 we identified that methods must be named the same as variables, i.e., the name can contain upper and lower case letters, numbers, and the underscore character, but may not start with a number. Unlike variable names however, method names do not need to be unique – you can define several methods with the same name, and this is known as method overloading.

We have in fact already been using method overloading. Consider the `WriteLine` method of the `Console` class, which actually has a large number of overloads, including:

```
void WriteLine(bool value)
void WriteLine(char value)
void WriteLine(decimal value)
void WriteLine(double value)
void WriteLine(int value)
void WriteLine(long value)
```



```
void WriteLine(float value)
void WriteLine(string value)
void WriteLine(uint value)
void WriteLine(ulong value)
void WriteLine(string format, object arg0)
```

You will notice above that there is an entry for most simple data types, as well as one of the overloads provided for composite formatting (Section 2.9).

To use method overloading, the compiler still needs to be able to determine which version of the overloaded method is being invoked, i.e., the method being invoked must be unambiguous. For this purpose, methods are distinguished by their signature (method signature), which must be different for each method with the same name. The method signature includes the number of parameters in the parameter list, the data types used, and how they are passed (input/reference/output parameters)¹³. At least one of these elements of the signature must be different for each method.

Method overloading can improve code readability, e.g., consider the following statements:

```
aNetworkConnection.SendInt32(123);
aNetworkConnection.SendFloat(1.23);
aNetworkConnection.SendString("123");
```

With method overloading the class defining the `aNetworkConnection` object could use one method name `Send`, and have the compiler determine the correct method to invoke/execute. This results in the following equivalent statements:

```
aNetworkConnection.Send(123);
aNetworkConnection.Send(1.23);
aNetworkConnection.Send("123");
```

The problem with this approach however, is that there is now a need for a new implementation/a new method overload for every data type that is to be supported. Failing to define a new method overload for every data type could lead to unexpected implicit conversions (Section 3.6) of data and/or errors in the program. A solution to this problem is generics, which we examine in Section 11.3.

3.4.3. Optional Parameters

Occasionally when developing software, there is a need two or more methods with nearly identical functionality. Consider the following examples:

```
if (enrolledStudents.Find(211123456) == true)
    Console.WriteLine("Student already enrolled");

if (enrolledStudents.Find(211123456, "SIT232") == true)
    Console.WriteLine("Student is enrolled in SIT232");

if (enrolledStudents.Find(211123456, "SIT232", Attempt.First) == true)
```

¹³ Some definitions of a method signature also include the return value type, however the C# programming language does not consider the return value type as part of the method signature for method overloading.

```
Console.WriteLine("Student is making first attempt at SIT232");
```

In order to implement the `Find` method indicated above, using the following definitions:

```
bool Find(int id)
{
    ...
}

bool Find(int id, string unit)
{
    ...
}

bool Find(int id, string unit, Attempt which)
{
    ...
}
```

Note: The parameter `Attempt` represents an enumerated data type

However an alternative approach is to use optional parameters¹⁴. Optional parameters are specified in a method definition like any other parameter, however when invoking the method it is not mandatory for the programmer to provide a value, i.e., the parameter is optional. To achieve this functionality, the parameter syntax is expanded as follows:

```
[ref|out] type name[ = default_value]
```

This new syntax allows the programmer when defining a method to specify a value to be used whenever the method is invoked without explicitly specifying that parameter (*default_value*).

Given this expanded syntax, we could write a single `Find` method which satisfies each of the above invocations, for example:

```
bool Find(int id, string unit = null, Attempt which = Attempt.Any)
{
    ...
}
```

In the above example, the `id` field is the only required parameter, whereas both the `unit` and `which` parameters are optional. If no value is specified for the `unit` parameter, it will contain a value of `null`. Similarly, if no value is specified for the `which` parameter, it will contain a value of `Attempt.Any`.

Following this approach, it is now possible to invoke the `Find` method with one parameter (`id`), two parameters (`id` and `unit`), or three parameters (`id`, `unit`, and `which`), satisfying the examples given above. However what is not apparent is whether it is possible to define the `Find` method to also permit only the `id` and `which` parameters to be specified, e.g.,

¹⁴ Optional parameters are also sometimes referred to as optional arguments.

```
if (enrolledStudents.Find(211123456, Attempt.First) == true)
    Console.WriteLine("Student is making their first attempt at a unit");
```

In fact, the `Find` method is already defined in a way which permits this, however the method must be invoked using named parameters, examined in Section 3.4.4.

3.4.4. Named Parameters

Until now when invoking a method we have only seen how to specify the parameters to be passed to the method in the same order they were defined, e.g., consider the following definition:

```
void someFunction(int paramOne, int paramTwo, int paramThree)
{
    ...
}
```

and matching example method invocation:

```
someFunction(valOne, valTwo, valThree);
```

These are known as positional parameters, i.e., a parameter (argument) that must be specified in the correct position. It is possible however to change the order in which parameters are specified by using named parameters¹⁵. Named parameters are used when invoking the method to specify which parameter is being passed using the following syntax:

parameter_name: parameter_value

Thus, to reverse the order of parameters passed, we change the example method invocation as follows:

```
someFunction(paramThree: valThree, paramTwo: valTwo, paramOne: valOne);
```

Importantly, it is also possible to combine positional parameter with named parameters, however once a named parameter is specified, the remaining parameters in the list must also be named, e.g., the following call is valid (one positional followed by one named):

```
someFunction(valOne, paramThree: valThree, paramTwo: valTwo);
```

however the following call is invalid (one named, one positional, one named), and will result in a syntax error:

```
someFunction(paramThree: valThree, valTwo, paramOne: valOne);
```

The syntax error occurs because once the second parameter is not named – after the first parameter was named, all remaining parameters must also be named.

¹⁵ Named parameters are also sometimes referred to as named arguments.

3.4.5. Recursive Methods

In Section 2.2.3 we examined three control structures for repetition (generically referred to as iteration): the while loop, do-while loop, and the for loop, and in Section 3.2.3 we have also examined the `foreach` loop. Another way of achieving repetition is known as recursion, whereby a method contains a call to itself (known as a recursive call). Although recursion is examined in some detail in the unit SIT221 Classes, Libraries and Algorithms, it is important to understand the basic concepts and the advantages and disadvantages of recursion.

Recursive algorithms are where a problem is not solved directly, but is reduced progressively until a problem with a known solution is reached, i.e., the algorithm results in the same problem to be solved, only smaller. One of the simplest examples of a recursive algorithm can be seen in the mathematical problem of factorial. The factorial of a number ($n!$) is the product of all values up to and including that number, e.g.,

$$5! = 5 * 4 * 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

$$3! = 3 * 2 * 1$$

$$2! = 2 * 1$$

$$1! = 1 * 1$$

$$0! = 1$$

Importantly in the above examples you will notice that the solution to each line also appears in the previous line, and the same examples could have equally have been written:

$$5! = 5 * 4!$$

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

or more generally as:

$$n! = n * (n - 1) !$$

Drawing upon this example, we can see that there are two elements needed for a recursive algorithm: the general case and the base case. The general case is what we see in the final formula – the expression of the solution in terms of a same problem, only smaller, e.g., to solve $5!$ we must first solve $4!$, to solve $4!$ we must first solve $3!$, and so on. The base case is then a known solution to the problem, which the general case progressively reduces the problem towards. For factorial, the base case is $0!$, which has a solution of 1.

The actual code that solves factorial recursively is shown in Figure 3.8. In this example, the `Factorial` method tests to see if the `number` parameter matches the base case, which if true the known solution is returned. Otherwise the general case is used

to find the solution, which is achieved by invoking the `Factorial` method again but with a value of `number - 1`, moving the problem one step closer towards the base case.

```
ulong Factorial(ulong number)
{
    ulong result;

    if (number == 0)
        result = 1;
    else
        result = number * Factorial(number - 1);

    return result;
}
```

Figure 3.8: Example recursive method for calculating Factorial

In summary, recursion provides an important alternative to iteration and most problems can be solved using either iterative or recursive approaches. Recursive solutions are often less efficient than their iterative equivalent, usually due to the information that must be maintained for each recursive method call – memory space may be allocated for each parameter, each variable declared in the method, any return value from the method, and any other temporary data generated during the execution of the method.

However it is important to understand that there are also problems which can be solved more efficiently using recursive solutions. Additionally, applying recursion to some problems results in a much simpler algorithm, and as such is less likely to contain errors (the more complex the code, the more likely there are errors). There are also data structures (used for storing data) that have a recursive structure, and as a result recursion algorithms often provide a more natural solution. Finally, recursion provides a very easy way to perform backtracking, which is very useful for algorithms such as path-finding (often used in games for example).

Task 3.1. Recursive Methods

Objective: The ability to develop recursive methods reliably is a difficult skill to achieve. Key to developing this skill is to gain a proper understanding of how the general case and base case work together. In this task you are presented with an unusual problem to be solved recursively.

Using Visual Studio, you are required to write a console application in C# that containing a recursive method named `SumUpTo()` that accepts a single parameter of type unsigned integer (`uint`) and calculates the total of all numbers (also a `uint`) up to and including that value, i.e., invoking `SumUpTo(5)` will return a value of `15` ($5 + 4 + 3 + 2 + 1 + 0 = 15$). Write a simple `Main()` that invokes this method based on user/console input and displays the result.

Example output (bold text represents input from user):

```
Please enter an integer value to sum up to: 10
```

The sum of all numbers up to and including 10 is 55

Hint: For the base case, use an input value of zero (0) which returns a result of zero (0). For the general case, you need to add the value of n to the sum of all numbers up to $n - 1$.

3.5. Scope

One of the fundamental concepts in programming is that of scope – the regions or sections of a program in which a name (variable name, method name, etc.) can be used without requiring some additional qualification, e.g., invoking a method as `SomeMethod()` is in scope, but not the same method invoked as `someObject.SomeMethod()`. Scope is important because you cannot use the same name twice in the same scope (unless overloading a method), e.g., you cannot create a variable and a property using exactly the same name in a class, which is why we usually prefix the variable with an underscore (`_`) or similar (discussed in Section 2.5).

Generally speaking, in C# the scope of a name is limited to the code block in which it was declared, whether that code block be for defining a class, defining a method, or for a selection or repetition control structure, however there are some exceptions. Here are some of the basic rules of scope for the C# programming language:

- ▮ A variable declared is in scope from the statement it is declared in until it reaches the end of the code block in which it is declared;
- ▮ A variable declared in the initialisation section of a for loop is in scope only for the statements associated with that for loop;
- ▮ A variable appearing in the parameter list for a method is in scope for the entire code block for that method;
- ▮ Member variables, properties, and methods are in scope for the entire class in which they are defined;

3.6. Argument Promotion and Casting

In Section 1.6, we considered how to convert data between different data types in the C# programming language. However by now you may have noticed that the compiler is inconsistently producing error messages when different data types are needed. Consider the following two variable declarations and initialisations:

```
int wholeNumber = 5.0 * 2;  
double realNumber = 5.0 * 2;
```

These two lines appear to be functionally identical, however the first of these two lines produces the compilation error: "Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)", however the second line compiles correctly. This (somewhat cryptic) error message is trying to tell us that we are somehow attempting to convert data that is of type `double` to the type `int`.

To explain this, start by considering the actual calculation being performed here. If you consider how literals are expressed in C# (Section 1.5), the above calculation

(`5.0 * 2`) involves two data types, a `double` (`5.0`) multiplied by an `int` (`2`). Neither the compiler nor the computer hardware knows how to multiply two different data types together, so the compiler automatically converts the `int` to a `double`, known as argument promotion.

Argument promotions are not done randomly, they follow a strict set of rules, as summarised in Figure 3.9. The figure shows each of the numeric simple types and with arrows between some of the types. Following the arrows, you will notice that it is possible to move from the `int` data type, through the `long` and `float` data types, into the `double` data type. Thus, an integer value (`int`) can be "promoted" to the `double` data type.

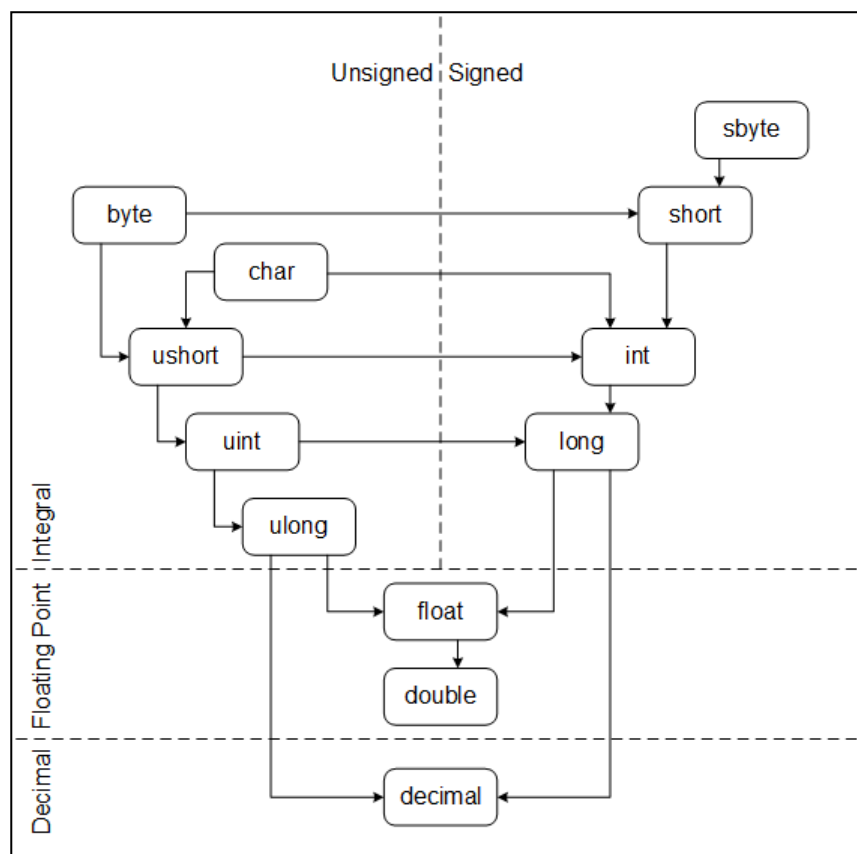


Figure 3.9: Argument Promotion in C#

Now that both of the operands to the multiplication operator are of the same type, the multiplication can be performed, producing a result which is also of type `double`. As the calculation is the same in both statements, we can now see that the above lines were equivalent to:

```
int wholeNumber = double_value;
double realNumber = double_value;
```

At this point, the error message indicated above should make sense. The second statement compiles correctly because assigning a `double` value to a variable of type `double` is straightforward. However, the first statement is attempting to assign a `double` value to a variable of type `int`, requiring the data to first be converted.

According to the rules laid out in Figure 3.10, no argument promotion is possible, thus an explicit conversion is required. We have already seen how to do this (Section 1.6), as follows:

```
int wholeNumber = Convert.ToInt32(5.0 * 2);
```

In this new version of the statement, the calculation still results in a `double` value, but it is converted to an `int` before it is assigned to the variable of type `int`. Another shorter alternative is to use casting, which uses the following syntax:

(type)expression

In the above syntax, *expression* represents a literal, variable, method call, or other expression that results in some value. The type that appears in parenthesis represents the data type that the result of the expression should be converted to. This results in the following statement:

```
int wholeNumber = (int)(5.0 * 2);
```

Importantly, note that casting is not a shorter version of the `Convert` class, there are differences. The most obvious difference is that casting does not support converting from a `string` and numeric types. More significantly, casting will not produce an error if data is lost in the conversion. For example, consider the following two statements:

```
int bigNumber = 123456;  
byte smallNumber = (byte)bigNumber;
```

From A, the byte data type only stores the values 0-255. Clearly, the value assigned to the variable `bigNumber`, 123456, cannot fit into the `smallNumber` variable, however no error message is produced. If the `Convert` class were used instead, an error would be produced in the form of an exception, which we examine in Section 9.3. Importantly, although the `Convert` class has this unique ability, we use casting for other purposes as well, most notably for Inheritance and Polymorphism, examined in Section 6.2.

3.7. Mathematics Support

The C# programming language is supported by the Microsoft.Net framework, a large collection of classes that provides a variety of functionality including console window input and output, GUI, file input/output, database access, web client and server, and so on. Many applications make use of mathematics functionality, which is found in the `Math` class. The `Math` class provides functionality for working with both the `double` data type and `decimal` data types. Some of the more useful members of the `Math` class include:

- | Constants:

- | `Math.PI`— The value of P;

- | Methods:

- | `Math.Abs`— Determine the absolute value of a number (eliminate any negative sign);
 - | `Math.Ceiling`— Round a value up to the nearest integer value;
 - | `Math.Floor`— Round a value down to the nearest integer value;

- | `Math.Max` – Determine the maximum of two numbers;
- | `Math.Min` – Determine the minimum of two numbers;
- | `Math.Pow` – Calculate the value of a number raised to some power, i.e.,;
- | `Math.Round` – Round a value to the nearest integer or nth decimal place;
- | `Math.Sqrt` – Calculate the square root of a number; and
- | `Math.Truncate` – Eliminate any decimals from a number.

Full documentation of the `Math` class can be found at:

<http://msdn.microsoft.com/en-us/library/system.math.aspx>

Task 3.2. Standard Deviation

Objective: Mathematical equations are used regularly throughout computing and as a professional software developer you will often be required to implement such equations. This task will help you to develop your skills working with loops, generic collections, and the mathematics functionality provided by Microsoft .Net to implement the standard deviation.

Using Visual Studio, write a console application in C# that accepts as input an arbitrary number of floating point values (type `double`), i.e., the user can enter any number of floating point values, and calculates the standard deviation of those numbers. The

$$\sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

formula for standard deviation is as follows:

Example output (bold represents input from user):

```
Please enter a numeric value or END to finish: 1
Please enter a numeric value or END to finish: 2
Please enter a numeric value or END to finish: 3
Please enter a numeric value or END to finish: 4
Please enter a numeric value or END to finish: 5
Please enter a numeric value or END to finish: END
The standard deviation of the entered numbers is 1.58113883008419
```

Note: If you do not understand the above formula, do a web search for "standard deviation". You will find many web sites explaining the formula in detail, which will help you to understand the relevant mathematics. You will also find alternative formulas for standard deviation, however your program must use the above formula.

Hint: Don't try to do all of the above functionality in one go – you'll make the task much harder than it actually is. The first stage you should aim for is to develop the loop which continuously prompts for data until the user types in 'END'. Build and run your program, making any changes to remove bugs, etc., until you are satisfied with that functionality. You can now rely on that functionality as you progress in the remaining tasks. The next stage might then be to capture the numbers and store them in a List object, displaying each of them

on the screen before the program terminates. Learning to approach solving programming problems in this manner, i.e., as a series of sub-problems, known as incremental development, is critical to becoming a successful programmer.

3.8. Validity Checking

When developing an application that interacts with the user, it is important to consider whether or not validity checking is necessary. Validity checking refers to the task of making certain that the information that the user enters is reasonable. The task of validating user input has become increasingly important and come under scrutiny in recent years as attackers have exploited weaknesses in validity checks. The general rule for validity checking is to assume that all input is unsafe, i.e., invalid, then only accept inputs that are known to be safe (by validating those inputs). In this section we cover a basic introduction to validity checking. More advanced validity checking is possible when regular expressions are used, examined in Section 10.3.

We consider only two simple checks at this time:

- | Make sure data input is the correct type, i.e., only digits (0-9), '-' (negative), or '.' (decimal place) are entered for a numeric value; and
- | Values should be in a correct range, both for the problem, e.g., the 30th and 31st of February do not exist, and appropriate for any data storage, e.g., storing 200 in an sbyte variable actually stores -56.

There are a number of static methods defined by the character data type (`char`) that are useful for this purpose:

- | `char.IsDigit`—tests for a decimal digit (0..9);
- | `char.IsLetter`— tests for a letter (a..zA..z);
- | `char.IsLetterOrDigit`— tests for either a letter or digit (a..zA..Z0..9);
- | `char.IsLower`—tests for a lowercase letter (a..z);
- | `char.IsNumber`— tests for a number (0..9 and also numbers in other character sets);
- | `char.IsPunctuation`— tests for punctuation (. , ! ? etc.);
- | `char.IsSeparator`— tests for a separator (space and new line);
- | `char.IsSymbol`—tests for symbols (\$+—etc.)
- | `char.IsUpper`— tests for an upper case letter (A..Z); and
- | `char.IsWhiteSpace`— tests for a white space character (space, tab, and new line).

Full documentation of the `System.Char` class (C# `char`) can be found at:

<http://msdn.microsoft.com/en-us/library/system.char.aspx>

Task 3.3. Validity Checking

Objective: Implementing correct validity checking for applications is becoming a fundamental requirement for any software developer. In this task we gain some early experience with validity checking.

Using Visual Studio, write a console application that accepts from the user four values and checks their validity: student ID number, year of birth, month of birth, and day of birth. The following rules must be applied for validity:

- | All values entered should contain only digits;
- | The student ID numbers should be either eight or nine digits long;
- | The year of birth must be a four digit year and be a minimum of 17 years ago and 100 years ago at most (the current year can be determined using the `DateTime.Now.Year` property);
- | The month should be in the range 1-12;
- | The day of birth should be in range for the appropriate month (assume February 29th is valid).

Your program should keep prompting the user for each of these pieces of data until they enter the correct value. Once all data has been entered correctly, it should be reproduced (written) on the console.

Note: Regular expressions may not be used for this task.

Hints:

- | To access each character in a string, use a foreach loop, e.g.,

```
string input = Console.ReadLine();
foreach(char c in input)
    // do something
```

- | To check if a character is a digit or not, use the `char.IsDigit()` method, e.g.,

```
if (char.IsDigit(c))
    // do something, it is a digit
else
    // do something else, it isn't a digit
```

Example output (bold text represents input from user):

```
Please enter the student ID: 21112345678
You have entered an invalid student ID, please try again.
Please enter the student ID: 214123456
Please enter the birth year: 2000
You have entered an invalid birth year, please try again.
Please enter the birth year: 1997
Please enter the birth month: 15
You have entered an invalid birth month, please try again.
Please enter the birth month: 1
Please enter the birth day: 50
You have entered an invalid birth month, please try again.
Please enter the birth day: 1
Validated data:
    Student ID: 214123456
    Birth date: 1/1/1997
```

Note: Starting code for this task can be obtained from the example code files provided in CloudDeakin.
