Session 1. Introduction

Welcome to the unit Object-Oriented Development. This unit continues the study of software development from the pre-requisite units and assumes that you have already gained an understanding and some experience in software development. In particular, we assume that you are familiar with designing and writing simple algorithms using the basic building blocks provided by programming languages including statements, decision structures (if statements, switch/case statements, etc.), looping structures (for loops, while loops, etc.), and the ability to develop code using a modular structure by applying methods (also known as functions).

In the first weeks of this unit, we will review the basic building blocks of the C# programming language while at the same time learning how to construct and work with objects. This will give you the time to either refresh your knowledge or to learn the syntax and basic library functionality of C# if you are new to the language.

Session Objectives

In this session, you will learn:

- The nature of an object-oriented application;
- What objects are, what they do, and how they are defined by classes;
- The basic structure and syntax of a C# console application; and
- How to write, build, and run applications written in C#.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*
  We begin introducing object-oriented concepts in this session focusing on the theory of abstraction and encapsulation.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  We will begin reviewing the basic concepts of the C# programming language in this session.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  We examine syntax errors and how to correct them in this session. General knowledge of the C# programming language is also 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 1

This chapter describes how computers work and the role of programming languages, C# in particular,, to allow you to develop an understanding of the context of our studies.

Recommended Reading: Chapter 2

This chapter provides a useful overview of the Visual Studio development environment. Students who completed the pre-requisite unit SIT102 at Deakin will already be familiar with Visual Studio and can safely skip this chapter. Alternatively, if you received credit for the pre-requisite and aren't familiar with Visual Studio should quickly review this chapter.

Required Reading: Chapter 3 up to, but not including, Section 3.9.

The C# programming language is introduced in this chapter and many of the basic building blocks used in constructing C# applications are covered.

Required Tasks

The following tasks are mandatory for this week:

- Task 1.1. Object Identification and Classification
- Task 1.4. Input and Output
- Task 1.5. Dealing with Syntax Errors

The other tasks in this section should be reviewed, and you should complete those tasks if you are not confident you have the required knowledge. Tasks that are not mentioned above do not need to be submitted however.

1.1. Introduction to Object-Oriented Development

Until now, our study of software development has focused on learning how to use, and gaining experience with, the basic building blocks of an application, e.g., variables, decision structures such as if statements and switch/case statements, looping structures such as for loops and while loops, and the use of methods (also known as functions). Developing a solid understanding of the basic building blocks, and how to combine them into useful algorithms, are critical skills for developing any application. However as the size of the application to be developed grows, it quickly becomes very difficult to see how these small building blocks will fit together to form the complete application. This is due to the complexity of such large applications.

To successfully develop complex applications, it is necessary to first decompose an application into much smaller problems that we either know the solution to, or can solve in a relatively short time. Object-Oriented Development, which we study in this unit, is the dominant methodology used for this purpose today. Other approaches include imperative programming (or procedural programming), functional programming, and logic programming, which we do not address here.

Object-oriented applications consist of a collection of objects that cooperate to achieve the required functionality of the application. Two key concepts exploited in this model are abstraction and encapsulation . Abstraction allows us to focus on those aspects of the problem that are important/relevant to the solution while ignoring those aspects that are irrelevant. Encapsulation, also known as information hiding or data hiding, maintains a separation between the external view of an object (its interface) and the implementation of that interface.

Exploiting the concepts of abstraction and encapsulation provides three main advantages. First, the application is very modular, and it is easy to exchange objects with new versions or different implementations of the same object. As long as the object's interface remains consistent, no changes are required to other parts of the application. Second, as the objects have clearly defined interfaces and functionality, the reusability of objects is high and they can often be re-purposed for new applications without modification. Third, once the overall architecture of the application is determined, i.e., what objects are required and how they will interact with other objects, the developer can focus on developing each different object separately, minimising the complexity of the development effort. Similarly, the development of different objects can be divided among separate developers and/or teams of developers for reducing overall development time.

For example, consider a motor vehicle. The abstraction/interface presented by the motor vehicle includes elements such as the steering wheel, accelerator pedal, break pedal, clutch pedal, and gear selector. The elements that are hidden/encapsulated include the engine, transmission system, exhaust system, and so on. Once you are able to drive a motor vehicle, you are able to exchange that motor vehicle for another, and you will be able to drive it as long as the same interface is maintained. Each motor vehicle can also be "reused" by different drivers as well. Similarly, the engine, transmission, exhaust system can be developed and tested by different engineering teams and can be used in more than one motor vehicle.

In an object-oriented application, an object can represent many different things, either real or virtual. Real objects could include a light switch, a student, a book, or a keyboard. Virtual objects could include an array, a queue, a text box, or an avatar (character) in a computer game. Each object has a collection of attributes and operations. Each attribute, represented by a variable, contains a single piece of data describing the object in some way, e.g., the state of a light switch (on/off), the ID number for a student (214123456), or the length of a queue (15 print jobs in queue). Each operation, represented by a method, defines the behaviour (actions and reactions) of an object, e.g., turning on the light, enrolling a student, or adding another print job to the queue. The combination of attributes and operations for a particular object defines the interface for that object.

As an example, an object might be used represent a textual display, such as a label on a window (virtual) or a display on a DVD player or similar (physical). Each of these objects could have attributes of what text is displayed, what colour/brightness the text

is, and so on. Operations could include the ability to change the text displayed, turn the display on/off, and so on.

When examining the objects in an application you will also notice that there are a number of very similar objects, e.g., many students can be enrolled in one unit, a company can be either a supplier or a client, books listed in a library catalogue or purchase order, and so on. We then group similar objects into a classification (class), where all objects of the same class have the same features, e.g., all person objects have a name, height, weight, age, eye colour, and so on. In practical terms, we develop an object-oriented application by writing classes, which we then use as templates for the creation of objects. Each object is then considered an instance of one of those classes. Class definitions consist of the variables, properties, and methods, which represent the attributes and operations of an object's interface.

Throughout this unit we will learn and apply the object-oriented methodology through the development of console applications, i.e., applications that run from a command prompt, in C#. As a programmer, you will be required to develop software in many environments, which could include windowed environments (Microsoft Windows, MacOS, X-Windows), console applications (Windows Command Prompt, MacOS Terminal and other Unix shell environments), mobile applications (smart phone, tablet), operating systems (drivers, services, etc.), and so on. Each of these platforms can have unique requirements and application programming interfaces (APIs), however the application of object-oriented development concepts does not change between them.

There are two main advantages to learning software development using console applications. First, these applications are important and prominent in a number of modern platforms, and it is important that you are able to develop for these platforms. Console applications are somewhat prevalent in Unix platforms, such as Linux, and throughout cloud computing (such as Amazon Web Services), and they are also become increasingly important for the Windows Server platform. Second, the development of an interface for a console application is usually much faster than for a GUI interface, allowing us to focus our time on learning object-oriented development. Moreover, if an application is implemented correctly, there should be a clear separation between the user interface and the data models and logic behind that interface, allowing the user interface to be replaced with an alternative interface at a later time (GUI interface or otherwise).

Task 1.1. Object Identification and Classification

Objective: This task introduces you to thinking about software development from the object-oriented perspective. This takes a while to get used to and it's important to start as quickly as possible. At such an early stage learning object-oriented development, you shouldn't expect to provide a perfect/great answer, so don't spend a long time on this.

Consider the following problem statement:

A friend of yours has decided to open a web store for technology products and has engaged you to develop the software to manage the sale and postage of products to customers. Your friend has told you that they intend to sell a variety of products including computers and software, entertainment systems (XBox, PlayStation, etc.), televisions, HiFi products (blu-ray players, PVRs, etc.), smart phones, tablets, and accessories (cables, etc.). Purchases can be made by credit card, PayPal, BPay, or Cash On Delivery. Delivery will be offered through a number of logistics companies including the regular postal service and various couriers (note that only some logistics companies provide Cash On Delivery options). The system is required to track both the suppliers (distributors and wholesalers) of the products for sale and the customers purchasing those products. The system must also track the accounts payable and receivable for suppliers/customers, and integrate with the business' clearing account and electronic payment gateway (for EFTPOS etc.) held with their bank.

Your tasks are to:

a. Identify the objects that are suggested by this problem statement. Note that nouns in a problem statement often suggest good objects.
b. Suggest classes that could be defined for these different objects.
c. Identify possible attributes and operations that could be relevant to/defined for each class.

Note: This task should not take a long time to complete, perhaps 10-15 minutes as a maximum. Approach the question as a brainstorming task, rather than something "to get right". You may also find it useful to write your answers to these questions in a text file (using Notepad or similar), before copying and pasting your answers into the quiz tool in CloudDeakin for submission.

1.2. Introduction to C#

The C# programming language is a modern object-oriented programming language originally designed at Microsoft and has since been standardised by both the European Computer Manufacturer's Association (ECMA) and by the International Organization for Standardization (ISO). These standards have been updated on several occasions since their initial release[1]. The core syntax of the C# programming language is common to many application programming languages including C, C++, and Java and compatible/similar syntaxes can be found in other languages including ECMAScript (JavaScript), ActionScript (Flash), Objective-C and Swift (Apple platforms), Python and Perl (scripting languages common in Unix/Cloud platforms), and so on. Collectively, these languages dominate the modern software development landscape. As such, it is important that you understand and be able to develop software using this syntax.

[1] The current state of the standards relevant to C# can be reviewed on the following web page: http://msdn.microsoft.com/en-us/vstudio/aa569283.aspx

```
/*************************************************************
** File: FirstProgram.cs
** Author/s: Justin Rough
** Description:
**     A simple program used to introduce the basic structure
**     of a C# application.
*************************************************************/
using System;

namespace FirstProgram
{
    class FirstProgram
    {
        // Main method, where the program's execution begins
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to OO Development!");
        }
    }
}
```

Figure 1.1: A Simple C# Application (FirstProgram.cs)

In Figure 1.1, a simple C# application is shown which only displays a message "Welcome to OO Development!". However, this program also demonstrates many important concepts. The following breakdown explains this simple program:

- Comments are a critical documentation tool for code, useful to both yourself and other programmers when reviewing code to fix bugs or extend the functionality of an application. Comments allow you to quickly understand the functionality expressed by the code the comments relate to. In this example there are two comments: a multi-line comment at the top of the file indicating its purpose, and a single-line comment indicating the purpose of the Main method. C# provides two types of comments:
  - Single-line comments, which begin with the character sequence `//` (two slashes) and continue until the end of a line; and
  - Multi-line comments, which begin with the character sequence `/*` (slash star/asterisk) and continue until the `*/` (star/asterisk slash) is found
- A namespace is identified for the definition of the new class called `FirstProgram`. We examine namespaces in , but for now use the name of your Visual Studio solution/project as the name of the namespace (Visual Studio will fill this in for you in most cases). The namespace is identified using the following structure:
  - The keyword `namespace` indicates that we are going to define one or more classes as part of a namespace, which is then followed by the name of the namespace, `FirstProgram`; and
  - The members of the namespace are then contained within a code block, which is enclosed by a set of braces (`{` and `}`).
- A class is defined called `FirstProgram`, which matches the file name the program is stored in (FirstProgram.cs). Although not strictly necessary, classes are normally stored in identically named files to make them easy to find later. Similarly each file usually contains only one class, although they can contain more than one (uncommon but sometimes used in more advanced programming). The class definition consists of the following elements:

    - The keyword `class` indicates that we are defining a class, which is followed by the name of the class, `FirstProgram`; and
    - The members of the class are contained within a code block, which is enclosed by a set of braces (`{` and `}`).
- A single method is defined, the Main method, which is a special method where the program starts its execution. The Main method definition consists of the following elements:
    - The method signature `static void Main(string[] args)` which C# identifies as the start of program execution; and
    - The body of the method is specified within a code block, which is enclosed by a set of braces (`{` and `}`). Code blocks are used regularly in C# to group together several related statements (lines of code).
- The Main method contains a single line of code which displays the message `"Welcome to OO Development!"` on the console. This is achieved by using the `WriteLine` method of the built-in `Console` class, which is used to output data to the console window (the command prompt window).

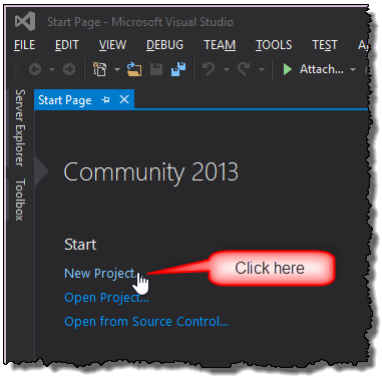
Task 1.2. Introduction to Visual Studio (Optional)

Objective: The objective of this exercise is to make sure all students are familiar with the Visual Studio development environment and are comfortable creating solutions and building software. This is critical if students are to begin learning object-oriented software development.

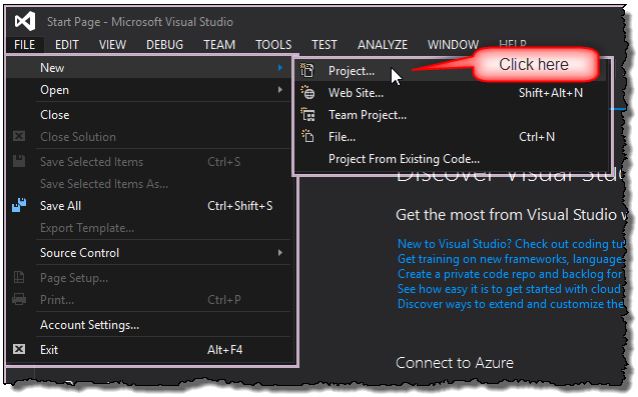
Using the code provided in Figure 1.1, you are required to create a project using Visual Studio and successfully build and run the program. Students who are familiar with Visual Studio from previous studies should be able to complete this task quickly on their own without the following instructions, and may choose to skip this task. Students who are using a development environment other than Visual Studio, or more than one environment, should ensure they could complete an equivalent task in all environments they have chosen to use.

To begin this task, start by opening Visual Studio on your computer. This is usually achieved through the start menu, by clicking the **Start** button, **All apps**, **Visual Studio 2013**, and click on the **Visual Studio 2013** program shortcut.

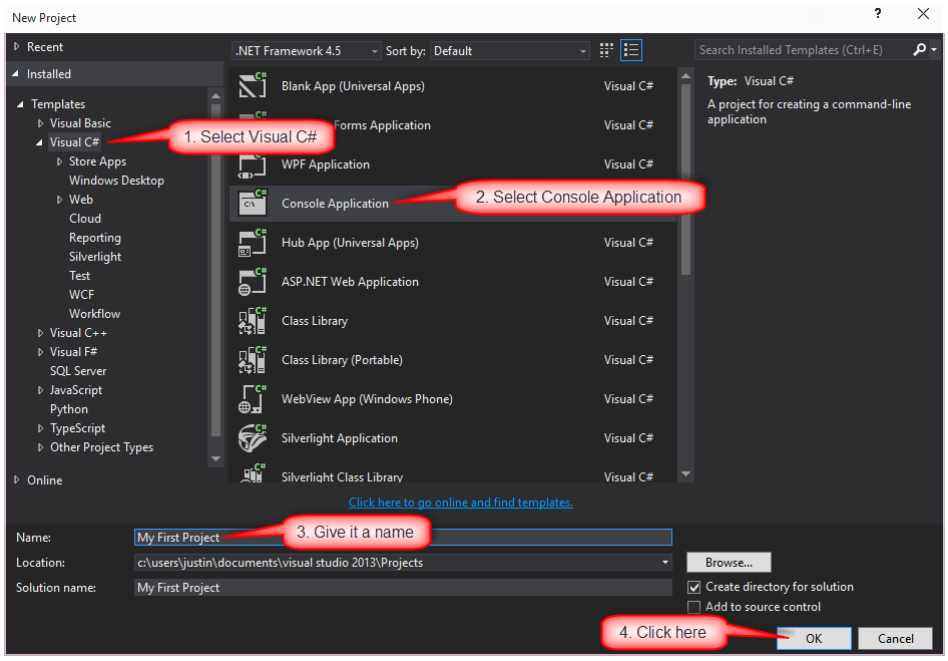
Once Visual Studio starts, we begin by creating a project. For our purposes, a project represents a single application that can contain any number of C# source code files (text files with a .cs extension). There are two ways to create a project, from the Start Page, or from the menu system. On the Start Page (usually displayed when Visual Studio is started, or click **View**, **Start Page**), there is a panel at the top left showing the list of Recent Projects, which also contains a link to create a project as shown (**New Project...**).



Alternatively, you can create a project by clicking on the **File** menu and then selecting the **New Project...** option.

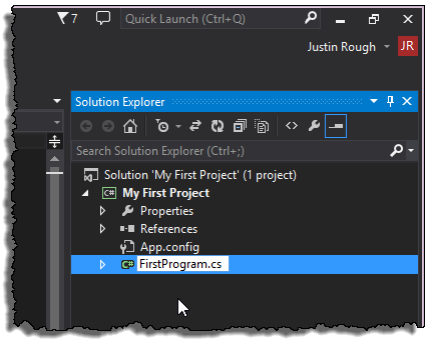


Each of the above actions opens the New Project dialog. The first step is to select the **Visual C#** Project type, then select the **Console Application** project template. Fill in the project's **Name**, which should resemble the name of the application or may be an abbreviated name representing the purpose of the project, e.g., Ass1 for the first assignment, Task_1.2 for a project representing this task.You may also wish to change the project's **Location**. Finally, click on the **OK** button to go ahead and

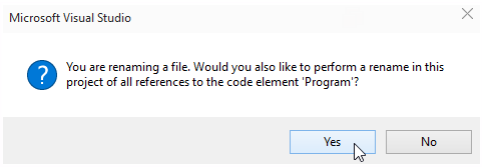


create the project. Visual Studio will now create the requested project.

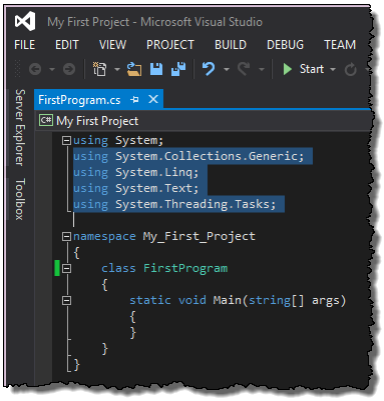
Once the project is created, Visual Studio will automatically creates a `class Program` and adds it to the project in a file named Program.cs. Often, you will not want to have a class Program in your application, and there is a need to rename the class. As discussed in Section 1.2, the file should be the same name as the class inside. Thankfully, Visual Studio makes this a simple change. If you rename a file using Visual Studio, it will automatically ask if you also want to change the file contained within. From Figure 1.1, we can see that the class should be named FirstProgram. In the **Solution Explorer** panel in Visual Studio, click the **Program.cs** filename to select it. After pausing for a moment, **click the file name again** and you will now be able to rename the file (alternatively **press the F2 key** on the keyboard to rename the file). Change the filename to **FirstProgram.cs** and then press the **ENTER** key.



Visual Studio will display a prompt asking if you are sure that you wish to rename the file, to which you should select **Yes**. You should now notice that both the filename and the class have the name FirstProgram.

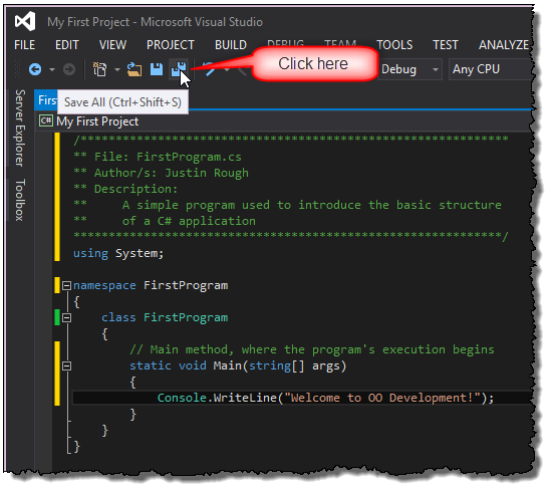


The first thing that you will notice about the source code that Visual Studio generates is that there a number of using statements that we aren't familiar with (highlighted in the figure). Some of these we will examine during the trimester, others are outside the scope of this unit. For the moment, these can just be erased from the file.

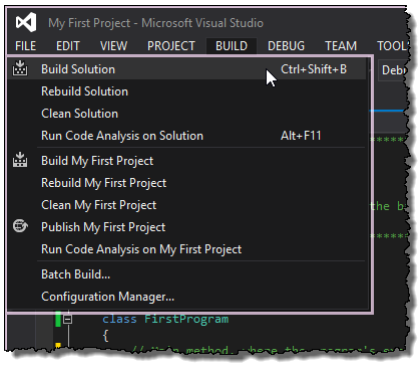


Continue editing the file until it matches the code in Figure 1.1. Feel free to use copy and paste to save time, but make sure you experiment with editing code in Visual Studio as it will provides you with a lot of assistance when coding, and you will want to quickly become familiar with this.

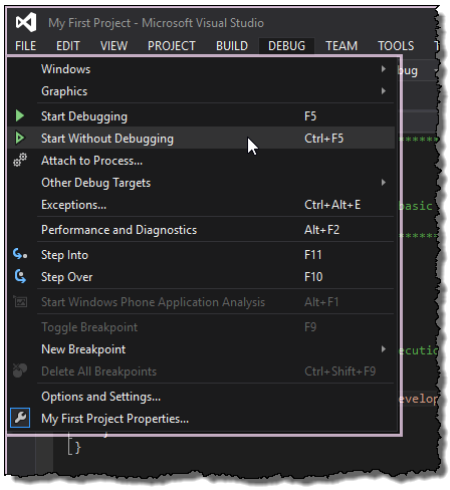
Save your project and the updated class file by clicking on the **Save All** button on the tool-bar.



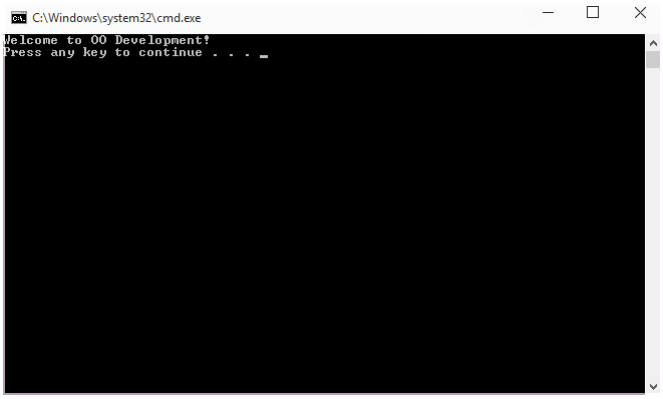
Build the project by clicking on the **Build** menu and then selecting the **Build Solution** option. This will compile your application and display any errors at the bottom of the screen.



Finally to run the application, click on the **Debug** menu and then select the **Start Without Debug-ging** option.



You should now see the resulting application output as shown to the right. Note that if the window only flashes before immediately disappearing, this is most likely because you selected the debugging option incorrectly in the previous step. Repeat the last step to get the window to display correctly if necessary.



If you have never submitted an assignment using CloudDeakin, or are a new student at Deakin, you should take the opportunity to practice using the assignment submission by submitting this task into the Practice Assignment Box in the Week 1 folder. Submission instructions for assignments are linked in the Assignments folders, please follow these instructions.

1.3. Statements, Indentation, and Readability in C#

Before we begin examining the C# programming language in detail, it is important to briefly discuss the structure of a statement in C#. A statement is the smallest unit of instruction defined by a programming language, i.e., one statement tells the computer to do one thing, or to perform one task. In some languages, such as Visual Basic.Net, a statement begins and ends on one line. In C#, a statement only ends when a semi-colon (;) is reached. If we the consider the example program in Figure 1.1 above, we see the following statement:

```
Console.WriteLine("Welcome to OO Development!");
```

You will notice the semi-colon (;) appearing at the end of this statement. Importantly, although this statement does in fact start and end on one line, it could easily have been broken using whitespace (spaces, tabs, and line breaks), which is ignored by the compiler. For example:

```
Console.WriteLine(
                              "Welcome to OO Development!"
            );
```

Functionally, there is no difference between these two statements. However it is immediately apparent that the second statement is taking up more space in the program code than the first statement. More importantly, if an entire program was written using the spacing of the second statement, the program would be very difficult to read and understand.

How easy the code of a program is to read is critically important. In general, the time taken to develop an application is substantially less than the amount of time the application will need to be maintained for. The maintenance of an application includes any bug fixes, improvements to the interface, extensions for new

functionality, etc., that occur over the lifetime of the program code after it is initially written. This results in a large amount of time spent by developers, both yourself and others, reading and understanding the code of the application. As such, when developing code, make sure you write it in a way that encourages/improves code readability.

Another major element that contributes to the readability of the code you write is how the code is indented (the indent style or indentation style), i.e., when a line/lines of code should be tabbed in another stop, and/or when to remove a tab stop. If you re-examine the above program (Figure 1.1), you will notice that any code following the start of a code block is indented one further tab stop (a code block begins with an open brace ('{')). At the end of the code block the tab stop is removed (code blocks end with a close brace ('}'). For example, consider again the Main method:

```
// Main method, where the program's execution begins
static void Main(string[] args)
{
    Console.WriteLine("Welcome to OO Development!");
}
```

Here you can clearly see the code inside the code block (between the braces) is tabbed in. This is the accepted indent style for this unit and you will need to quickly become used to programming with this style. By applying this indentation style to all of your code, you will also find that your code is generally more readable. Note that this is the most popular indent style in use today, however there are a number of alternative indentation styles[2]. The editor in Visual Studio, and many other programming editors, will assist you to use this indenting style in your own code but can be modified to support other indenting styles.

1.4. Variables

When writing software, you will regularly need to store some data, whether that data is provided by the user, the system, or is used to control the execution of your program. For this purpose, we use the memory of the computer. In general, memory provides a vast number of bytes that we can allocate to store data, which we allocate by creating (or declaring) a variable, each of which represents a single location in memory for storing data. The memory of a computer has no real concept of the type of data (data type) we want to store, e.g., numbers, letters, symbols, and so on, thus most programming languages including C# require programmers to identify the type of data to be stored[3]. The C# syntax to declare a variable is as follows:

type *name1*[= *value*][, *name2*[= *value*][, ...]];

[2] For further information on indentation styles, see http://en.wikipedia.org/wiki/Indent_style

[3] It is worth noting that C# does in fact allow you to create variables without specifying the data type through the use of the keyword 'var', however these should only be used when writing code that uses "anonymous types" which are beyond the scope of this unit. As such, use of the 'var' keyword is not used in, or permitted in assessment for, this unit.

In the above syntax, *type* represents the type of data that will be stored in the variable. A single piece of data is stored in a simple data type, also known as a fundamental data type or primitive data type. A data type that stores more than one piece of data is known as a complex data type, composed of one or more variables (of simple or complex types). We examine complex data types later in the unit, beginning with coverage of classes in Section 2.3.

After the data type is specified, one or more variables can be declared. Each variable name may optionally be followed by the specification of an initial value for the variable by inserting an equals symbol (`=`) followed by the value used to initialise the variable (known as variable initialisation). The listed variables (and any matching initial values) are then separated by commas (`,`) and the statement is terminated with a semi-colon (`;`). There are a large number of simple types provided by C#:

- Signed integrals (whole numbers): `sbyte`, `short`, `int`, `long`
- Unsigned integrals (positive whole numbers): `byte`, `ushort`, `uint`, `ulong`
- Floating point: `float`, `double`
- High-precision decimal: `decimal`
- Boolean: `bool`
- Unicode character: `char`

A complete list of simple data types, and the range of values they can store, can be found in A. For example, to declare a variable to store the year of someone's birth, e.g., 1995, we need an integral (a whole number without decimals) capable of storing up to four digits, but no negatives are required making unsigned variables appropriate. Examining the above list and A, we see that there are four options which can store different values:

- `byte` – with a range of 0 to 255, is not adequate to store a year[4];
- `ushort` – with a range of 0 to 65,535 is more than adequate to store a year;
- `uint` – with a range of 0 to 4,294,967,295, would store a year but is in excess of our requirements; and
- `ulong` – with a range of 0 to 18,446,744,073,709,551,615, is even further in excess of our requirements.

In addition, there is also the `string` data type, which is used to store a sequence of characters or text (i.e., a string of characters or text)[5].

The final requirement before writing a variable declaration is to determine the name of the variable. Variable names, like other identifiers in C# (method names, class names, structure names, etc.), can consist of upper-case letters, lower-case letters, the underscore character (_), and numbers. Note however that identifiers cannot begin with a number. The most important thing when deciding on a variable name is

[4] Although it might be possible to store only the last two digits of the year (95 for the above example), this should be avoided in general. Storing the year as only two digits, and the associated calculations and comparisons on a two digit year, was a major factor in what was widely known as the Y2K bug. See http://en.wikipedia.org/wiki/Y2k for further information.

[5] Note that the string data type is in fact not a simple data type, however it behaves in the same manner as a simple data type and can be treated as such for the time being. We will examine strings later in Section 10.2

to use a name that is representative of the data to be stored. For our example, an obvious variable name would be birthYear, resulting in a declaration as follows:

```
ushort birthYear;
```

The variable name above demonstrates what is known as Camel Case, which is used regularly in modern programming. Camel Case refers to the joining of multiple words together without spaces or underscores, with the first letter of every word, except for the first word, being capitalised. A variation of Camel Case, often referred to as Pascal Case, is where the first letter of the first word is also capitalised. In this unit we will use Camel Case for names of local variables and method parameters, and Pascal Case for naming attributes, properties, methods, and classes. Another important style of variable naming that you will encounter in a career involving programming is Hungarian notation. Hungarian notation was particularly popular throughout the 1990s and is still used in some situations today, particularly for GUI programming. Variables named using Hungarian notation include an indication of the data in the variable name, e.g., a `string` named `FirstName` becomes `strFirstName`. Hungarian notation is not recommended for use in this unit, however you may use it if you wish. More information on Hungarian notation is provided in B, and a general discussion of variable naming can be found at:

http://en.wikipedia.org/wiki/Naming_conventions_(programming)

Task 1.3. Declaring Variables (Optional)

Objective: The objective of this exercise is for you to practice two fundamental skills for developing C# applications: (1) writing statements, and (2) being able to select an appropriate data type.

For the following pieces of data, write a variable declaration in C# selecting the most appropriate data type and an appropriate variable name:

- The day of the month, e.g., 5
- The salary of an employee, e.g., $68,000
- The price of an item for sale, e.g., $24.95
- The value of PI (3.1415926535897932384...)
- The grade received for a unit (e.g., C for credit)

Note: The purpose of this task is to select the most appropriate variable type, not just any data type that would work, e.g., an integer would not be the most appropriate for the day of the month, even though it would work.

1.5. Literals

Closely related to the use of a variable is the use of a literal. Literals represent values that can be assigned to variables, used in calculations, used in method calls, and so on. In particular, literals are the name given to actual values appearing in a program's code.

- Integral types
  - `int` – specified by a number containing one or more digits in the range 0-9, e.g., `55`;
  - `uint` – specified the same as for an `int`, with the addition of the letter `U` appended (upper or lower case), e.g., `55U` or `55u`;
  - `long` – specified the same as for an `int`, with the addition of the letter `L` appended (upper or lower case[6]), e.g., `55L` or `55l`;
  - `ulong` - specified the same as for an `int`, with the addition of the letters `U` and `L` appended (upper or lower case, in any order), e.g., `55UL`, `55ul`, `55Ul`, `55uL`, `55LU`, `55lu`, `55lU`, or `55Lu`;
- Floating point types
  - `float` – specified by a number containing one or more digits in the range 0-9, optionally with a fractional part specified after decimal place (.), with the addition of the letter `F` appended (upper or lower case), e.g., `5.5F` or `5.5f`;
  - `double` – specified the same as a float, except without the letter `F`, and optionally with the letter `D` appended (upper or lower case), e.g., `5.5`, `5.5D`, `5.5d`;
- Other types
  - `bool` – `true` or `false`;
  - `char` – specified by the actual character appearing between apostrophes, e.g., `'X'`;
  - `decimal` – similar to a floating point type, the decimal type stores numbers with a fractional part (the decimal type stores fractional numbers more accurately than floating point types but is not efficient for mathematical calculations), specified the same as a float except with the letter `M` appended (upper or lower case), e.g., `5.5M` or `5.5m`; and
  - `string`[7] – specified by the desired sequence of characters appearing between double quotes, e.g., `"Welcome to OO Development!"`.

1.6. Data Type Conversion

There is often a need to change how data is represented in memory or how the data will be interpreted by the computer, i.e., to change from one data type to another. This is particularly common when you consider that all data entered by a user is in a textual format, either a single character, or a string. In this session we consider how to convert data from a string to one of the simple types identified above in Section 1.4.

Conversion is possible using the utility class `Convert` provided by Microsoft.Net. The syntax for a call to the conversion utility is as follows:

```
Convert.Totype(source);
```

[6] For readability, always use the upper case `L` in preference to the lower case `l`, which is hard to distinguish from the digit `1`.

[7] It is not uncommon for someone learning C# to get confused between the types `string` and `String` in C# (note the difference in the capital letter). In general, unless the code you are writing is something very unusual, you should be using the `string` type (all lower case). The same issue occurs for the `double` and `Double`, `char` and `Char`, and `decimal` and `Decimal` data types. The issue does not occur with other types as they have different names, e.g., `int` versus `Int32`.

In the above syntax, *source* represents the data that you want to convert, and *type* represents the (Microsoft.Net) type you want to convert to. Here is an example of declaring a `string` containing the value `"55"` and converting it to an `int` type (known to `Convert` by its Microsoft.Net data type `Int32`):

```
string sourceData = "55";
int destinationData = Convert.ToInt32(sourceData);
```

A full list of `Convert` methods for each of the simple data types is listed in B.

1.7. Obtaining Input and Formatting Output

Interacting with the user requires some form of input from, and output to, the user interface that the user is running your program from. When a GUI application is used, input is usually in the form of text boxes, radio/check buttons, list boxes, and so on. Output is sent to the user using labels, graphics, changing the contents of other controls, displaying additional windows/dialogs, and so on. For a console application, which does not contain these GUI elements, input and output is always in the form of text, which is either entered by the user or displayed for the user to read.

Where information is to be obtained from the user in a trivial format, e.g., only a name, a student ID, a measurement, and so on, input is obtained simply in C# using the `Console.ReadLine()` call, which returns a `string` result, for example:

```
string userInput = Console.ReadLine();
```

In the above example, whatever is typed in by the user will be stored in the variable named `userInput`, which can then be converted to other data types as needed using the data conversion discussed in Section 1.6. For more complex input we will examine regular expressions in Section 10.3.

When obtaining input from the user, it is important to first notify them that the program is expecting to receive their input by displaying a message. This is known as prompting the user, and the text that is displayed is known as the prompt, i.e., a short message which indicates to the user that they are expected to enter a response. Displaying a prompt is a very simple example of output. Output is sent to the user using one of two calls, either `Console.Write()` or `Console.WriteLine()`. Each of these methods takes a parameter for the data (simple data type) to be sent to the user. The difference between the two, is that `Console.WriteLine()` will move to the next line on the terminal, as shown in Figure 1.2.

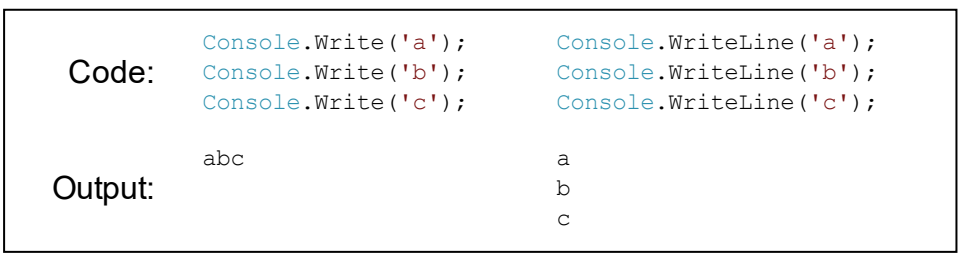


Figure 1.2: Output from `Console.Write()` and `Console.WriteLine()` Methods

Prompts can take many forms, in the same way as you could ask someone a question in many different ways. Consider the following examples:

```
What is your name?
Please enter your name:
Name>
```

Each of these prompts are requesting the same information from the user and could be used in any application. The nature of the application may result in one style of prompt being more appropriate than the other/s, or a program may even use more than one style of prompting. The following points should be followed when writing prompts to minimise confusion in the user:

- Clearly indicate what input is required;
- Clearly indicate the format required if relevant, e.g., for date or time entry;
- Be consistent, i.e., if asking for the same information many times, for the same reason, use the same prompt;

Here is a complete example for obtaining a student ID from the user, and converting it to an `int` type:

```
Console.Write("Please enter your student ID: ");
string input = Console.ReadLine();
int studentID = Convert.ToInt32(input);
```

Obviously, prompting the user for information is not the only reason to send output to the user. Information can be sent to the user in the same manner as discussed above, or you may also choose to apply some formatting to the data. Consider the example code in Figure 1.3. This example demonstrates a number of new concepts. Firstly, the program provides a more complete example for prompting the user and retrieving information using different data types. In particular, the program retrieves the user's given name (`string`), family name (`string`), student ID (`int`), and age (`byte`).

```
/*************************************************************
** File: ConsoleIO.cs
** Author/s: Justin Rough
** Description:
**     A simple program demonstrating how to prompt the user
**     for data, which is then output back to the user using
**     formatted output.
*************************************************************/
using System;

namespace ConsoleIO
{
    class ConsoleIO
    {
        static void Main(string[] args)
        {
            Console.Write("Please enter your given name: ");
            string givenName = Console.ReadLine();
```

```
            Console.Write("Please enter your family name: ");
            string familyName = Console.ReadLine();
            Console.Write("Please enter your student ID: ");
            string input = Console.ReadLine();
            int studentID = Convert.ToInt32(input);
            Console.Write("Please enter your age: ");
            // note variable input is declared above
            input = Console.ReadLine();
            byte age = Convert.ToByte(input);

            Console.WriteLine("\nID {0}: {1}, {2} ({3} years old)",
                              studentID, familyName, givenName, age);
        }
    }
}
```

Figure 1.3: Example of Input, Formatted Output, and Data Conversion in C# (ConsoleIO.cs)

Once the information is retrieved from the user, a summary line is displayed, e.g.,

```
ID 211123456: Smith, Jo (18 years of age)
```

This is a simple example of what is known as formatted output. The first parameter to the `Console.WriteLine()` call is known as the format string. The format string determines how the actual text should be displayed and contains a number of format items which are replaced with the values retrieved from the user. There are four format items in the format string: `{0}`, `{1}`, `{2}`, and `{3}`. Notice that the syntax for a format item is a number contained within braces (`{}`). The number refers to which parameter appearing after the format string should be used, with the zeroth parameter being the first parameter appearing after the format string, i.e., `{0}` refers to the variable `studentID`, `{1}` to `familyName`, `{2}` to `givenName`, and `{3}` to `age`. In addition to the format items, the first two characters in the control string (`\n`) represent one of a number of possible character escape sequences. An escape sequence is a sequence of characters, usually beginning with a back-slash character (`\`) followed by one or more other characters, which can represent:

- Elements that do not have a physical appearance, i.e., not an alphanumeric character or symbol;
- Elements that cannot be represented due to conflicts with syntax, e.g., given that a string in C# is surrounded by double quotes (`""`), how do you output a double quote character?; and
- The back-slash character itself.

The most commonly used escape sequences are the new line character used in the above example (`\n`), a tab character (`\t`) useful for displaying information in a table; the single quote (`\'`) and double quote (`\"`) characters, and the back-slash itself (`\\`). A more complete list of escape sequences is provided in C.

Note: We will reconsider formatted output in Section 2.9 where we examine mechanisms for more advanced formatted output. For now, complete the task below using only the mechanisms described above.

Task 1.4. Input and Output

Objective: Obtaining input from the user and producing formatted output (for any textual output) are two skills required regularly throughout a programming career. Formatted output is also an important skill for generating printed reports as well, which we will examine in detail later in the trimester. In this task you will gain experience with both skills.

For this task you are required to obtain from the user the following information: family name, given names, title (Mr, Mrs, Ms, Miss, etc.), country of birth, and spoken languages. Use a `string` variable to store each piece of data. Display a report on the screen containing this information, matching the following format:

```
**********************************************
     Field      Value
**********************************************
      Name:     The Hon Tony Abbott
   Born In:     United Kingdom
    Speaks:     Politics, English
**********************************************
```

For the above report, the following information was entered: "Abbott" (family name), "Tony" (given names), "The Hon" (title), "United Kingdom" (country of birth), and "Politics, English" (spoken languages). Note that only spaces and the tab (`\t`) escape sequence were used to create the above output.

1.8. Arithmetic Expressions

A fundamental skill of any programmer is the ability to write an expression. The most common use of expressions are for performing arithmetic (mathematical calculations), and for conditions (for making decisions, terminating loops, and so on). Simple expressions are constructed through the use of an operator and one or more operands. Operators represent arithmetic operators such as `+`, `-`, `*`, and `/`. Operands represent the parameters to the operator, e.g., in the expression `a + b`, the variables `a` and `b` are operands to the operator `+`. Note that a literal can replace one or both of the variables in such an expression, e.g., `a + 5`, `5 + b`, or `5 + 5`.

There are three types of operators: unary operators, binary operators, and ternary operators. Put simply, unary operators are operators requiring only one operand, e.g., `-a` (negative), binary operators require two operands, e.g., `a + b` (addition), and ternary operators require three operands[8].

An arithmetic expression is an expression that is used to perform very simple mathematical calculations, i.e., arithmetic. There are five relevant operators available for arithmetic: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulus, or remainder after division). Each of these are binary operators and can be expressed

[8] The C# Programming Language only supports one ternary operator, the conditional operator (`?:`), which we examine in Section 2.2.2.

either as a single calculation, e.g., `a * b`, or can be written as a sequence of operations, e.g., `a / b * c`. Any expression containing two or more operators is known as a compound expression.

Importantly C# follows the rules of mathematics when evaluating arithmetic expressions. For example, consider how to calculate the result for the expression:

```
8 / 2 * 4
```

This example appears to have two possible answers. If the division were to be performed first, the answer is then `4 * 4 = 16`. Alternatively if the multiplication were to be performed first, the answer would instead be `8 / 8 = 1`. The mathematical rules for arithmetic dictate that the division should be performed first. Most students learn the rules for arithmetic with the help of the BOMDAS or BODMAS acronyms. The rules tell us (among other things) that multiplication and division are always performed before addition and subtraction. Multiplication and division have equal priority and are evaluated from left-to-right. Similarly, addition and subtraction have equal priority and are evaluated from left-to-right.

For programming, these rules are expressed as precedence and associativity. Precedence defines the priority of an operation, where the operators with the highest priority (highest precedence) are performed first, followed by the operators with the next highest precedence. When two or more operators appear together in an expression that have equal precedence, they are evaluated according to their associativity. Operators can either be left associative, where operators are performed from left-to-right, or right associative, where operators are performed from right-to-left. For the five operators considered in this section, the `*`, `/`, and `%` operators have equal precedence, and are of higher precedence than the `+` and `-` operators, which also have equal precedence. All of these operators are left associative.

It is sometimes desirable to override the rules of precedence and associativity, which is achieved by introducing parenthesis into an expression (`(` and `)`) to bracket those sections that should be performed first, e.g., `8 / (2 * 4)`. The expression in the parenthesis is evaluated first as parenthesis have (equal) highest priority[9]. Parenthesis can also be used in an expression to clarify the order in which operators will be evaluated. This is useful to improve the readability of your code when dealing with an unusually complex expression.

Arithmetic calculations are much more useful if the result of those calculations can be stored for later use. For this purpose, we use variables and the assignment operator. Consider the following examples:

```
// Example 1
int result;
result = 8 / 2 * 4;
```

[9] Parenthesis have equal highest priority with a number of other operators (see Appendix E). The operators with equal precedence will not impact the vast majority of the expressions you will write, and usually only need to be considered in more advanced programming.

```
// Example 2
int result = 8 / 2 * 4;

// Example 3
int a = 8, b = 2, c = 4;
int result = a / b * c;
```

The three examples shown above are equivalent. The first example declares the integer variable `result` on the first line, then assigns the result of the expression on the second line. In the second example, these two lines are combined together into a single line using variable initialisation (the variable is initialised to, or given an initial value of, the result of the expression). The third example demonstrates how variables can be used instead of literals in the expression. Note that even though literals have been used in the above examples, we could equally have retrieved the values from the user at the console as per Section 1.7, above.

Exploiting the concepts we have covered so far in this section we can write a simple program to calculate the GST payable on an purchased item[10]. This program is shown in Figure 1.4, and also demonstrates two new concepts. First, you will notice that it is possible to perform rounding of floating point types, including the decimal type, using the `Math.Round()` method. The first parameter is the value to be rounded, the second parameter (optional) indicates how many decimal places the number should be rounded to. Second, you will notice that the format items in the `Console.WriteLine()` statements have a '`:c`' appearing after the format item number. This tells C# that the values are not just numbers, but they are currency values (monetary values). Provided with this information, C# will use the settings in the operating system to display the currency value correctly[11]. Note that it is not important that you understand these concepts yet, but feel free to experiment with them.

```
/*************************************************************
** File: GSTCalculator.cs
** Author/s: Justin Rough
**     A simple program demonstrating arithmetic expressions
**     applied to user input for the calculation of GST in
**     Australia (equal to 10% added to the sales price).
*************************************************************/
using System;

namespace GSTPayable
{
    class GSTCalculator
    {
        static void Main(string[] args)
        {
            Console.Write("Please enter the GST-ex price: $");
            decimal gstEx = Convert.ToDecimal(Console.ReadLine());

            decimal gstPayable = gstEx * 0.1M;
```

[10] GST in Australia is a flat 10% consumption tax applied to goods and services, with some exclusions such as for unprocessed food. For non-Australian citizens, this can sometimes be confusing, e.g., a raw/frozen chicken is GST free, however a cooked chicken (processed) attracts the GST.

[11] Currency settings can be found in the Regional settings applet in the Control Panel.

```
            decimal gstInc = Math.Round(gstEx + gstPayable, 2);

            Console.WriteLine("Total GST payable on {0:c} is {1:c}", gstEx, gstPay-
able);
            Console.WriteLine("GST inclusive price is {0:c}", gstInc);
        }
    }
}
```

Figure 1.4: Example of Arithmetic Expressions using User Input in C#

1.9. Syntax Errors

Every programming language defines a syntax through which a programmer specifies the functionality of their programs. The syntax of a programming language can refer to a number of elements, such as:

- The format of a statement;
- Mathematical expressions;
- Control structures (`if`, `for`, `while`, etc.);
- Function definition and invocation;
- Class definition and instantiation (object creation);
- An so on.

The rules of a programming language's syntax is roughly equivalent to the words and grammar of a spoken language. When compiling a program that you have written, the compiler reads the code that you have written and tries to interpret the code according to these rules. If the code does not match the rules, the compiler reports what is known as a syntax error. When a syntax error is identified, the compiler will produce a message indicating the following:

- Which source code file (.cs file) the error was found in;
- The line of the file where the error was found[12]; and
- A general message describing what the compiler knows about the error.

After a syntax error message is output, rather than stop the compilation process, the compiler tries to keep going. This can often mean that further errors are found, not necessarily because there are more errors, but because the first error caused the compiler to miss some information. For example, if a compiler cannot understand a line of code where you have declared a variable, a syntax error may then be generated for every line of code that you referred to that variable as the compiler no longer recognises it. Thus, fixing the first error causing the declaration to fail will in turn fix these remaining errors. As a general approach, always focus on fixing the first error listed by the compiler, then try to compile again before looking at the next error. In time, you will begin to understand the error messages better, and how they relate to each other, and

[12] It is important to note that compilers generally start at the beginning of a source code file and process the instructions contained in that file one line at a time. This means that sometimes a compiler will not find a syntax error until several lines after the syntax error occurred, thus if you can't see an error on the line indicated by the compiler, be sure to check the lines of code that appear immediately above the line indicated by the compiler as the error may have occurred earlier.

you will slowly learn when you need to recompile your code after fixing only one or after fixing several error messages[13].

Finally, when compiling a program a compiler will sometimes notice some strange or unusual code, for which it outputs a warning message. Warning messages do not stop a program from compiling, however it is important to check these warning messages as they are often indications of bigger problems in the code – the final version of a program should not generate any warnings when compiled.

The C# compiler can produce hundreds of different error messages[14]. Learning what each of these error messages will take time and initially the messages will seem very cryptic. Once you have gained some experience and a deeper understanding of how programming languages work however, you will begin to learn exactly what an error message is telling you, even reaching the point where you will often know what is wrong with the code before you have even seen it.

Task 1.5. Dealing with Syntax Errors

When writing code for an application you will often experience syntax errors. The ability to quickly identify and correct syntax errors is a critical skill for every software developer.

The following program contains 13 errors, most of which are syntax errors. Two errors are identical, and another two errors are very similar. Your task is to correct these errors to allow the program to compile and run correctly.

```
/************************************************************
** File: ThreeNumbers.cs
** Author/s: Justin Rough
** Description:
**     A simple program that calculates the sum of three
**     numbers and what percentage/factor of the sum the
**     three numbers contribute.  This program is provided
**     with many syntax errors as an exercise in correcting
**     such errors.
************************************************************/
using System;

class ThreeNumbers
{
    static void Main()
    {
        string input

        Console.Write("Enter the first number: ");
        input = Console.ReadLine();
        int num1 = Convert.ToInt32(input);
```

[13] Note that modern versions of Visual Studio also check your code continuously as you correct syntax errors, updating the list of errors found. This feature however should not be relied upon, as it can often get confused and will not always be consistent with the current state of the program code.

[14] See http://msdn.microsoft.com/en-us/library/ms228296(v=vs.120).aspx for a list of error messages that can be produced by the Microsoft Visual C# compiler.

```
            Console.Write("Enter the second number: );
            Console.ReadLine(input);
            int num2 = Convert.ToInt32(input);

            Console.Write("Enter the third number: ");
            input = ConsoleReadLine();
            int num2 = Convert.ToInt32(input);

            Console.WriteLine();

            int sum = num1 + num2 + num3;
            Console.WritLine("The sum of the numbers is {0}", sum);

            Console.WriteLine();

            float factor = (Convert.ToSingle(num1) / sum) * 100;
            Console.WriteLine("The number {0} represents {1,3:f}% of the sum",
num1, factor);
            float factor = (Convert.ToSingle(num2) / sum) 100f;
            Console.WriteLine("The number {0} represents {1:f,3}% of the sum",
num2, factor);
            float factor = (Convert.ToSingle(num3) / sum * 100f;
            Console.WriteLine("The number {0} represents {1,3:f%} of the sum",
num3, factor);
        }
}
```

Note: The code for this task should be obtained from the example code files provided in CloudDeakin, not by copying and pasting from this page.