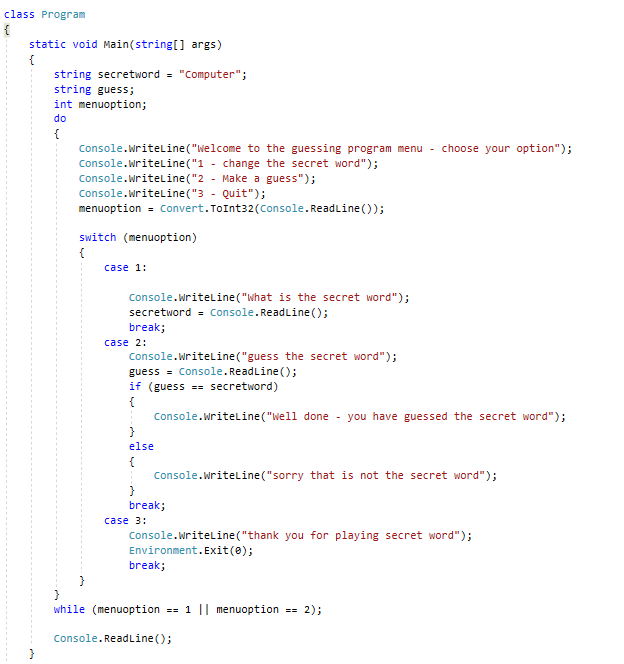
# Chapter 6 – Subprocedures and Functions

## The Static Void Main () procedure

In a console mode program, execution of the program will start here. The computer program below stores a secret word and then allows the user to either guess the word, change the secret word or quit the program.

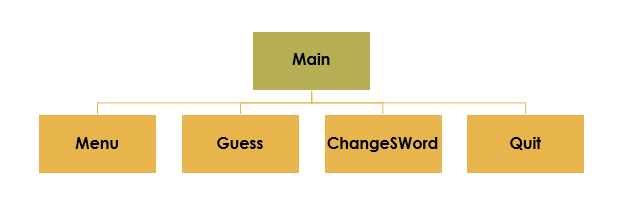
### The secret word program



So far, all the programming we have done (in the other chapters) has placed code inside the Static Void Main () procedure. However, it is more common (and more efficient) in programming to break down programs into smaller chunks with individual procedures performing single tasks. So, our secret word program could be decomposed into four separate tasks:

1. outputting the menu
2. making a word guess
3. changing the secret word
4. quitting the program

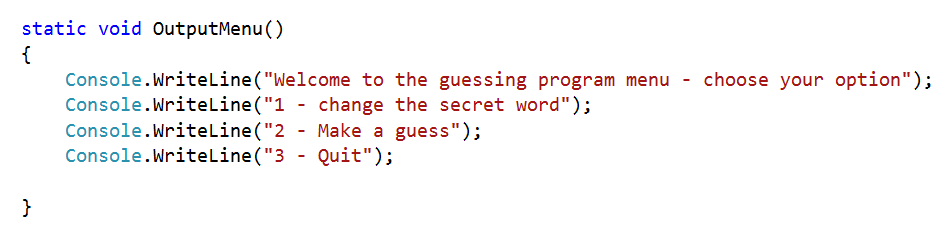
### Top-down diagram to show how the secret word program could be decomposed



## Writing a procedure that does not return a value

A procedure is a block of code that will perform a single task or a set of related tasks that can be called by another procedure (like the static void Main ()).

The procedure below will **output the menu** for the secret word program.



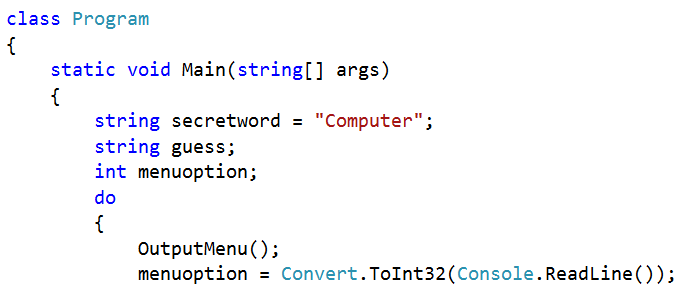
A procedure sometimes performs the desired operations without returning a value to the calling procedure. Such methods have a return type **void**. In this case, it cannot be called as part of an assignment statement.

NOTE: in C# all procedure identifier names must begin with a **capital letter**.

For this procedure to be executed it needs to be **CALLED** by another procedure.

## How to CALL a procedure

The **OutputMenu()** procedure from above is **called** in the **Main()** procedure. The program will execute the **OutputMenu()** procedure and then control will be passed back to the **Main()** procedure when the execution of the **OutputMenu()** procedure is complete.



In order to call the procedure to execute, we need to state the **identifier** followed by a set of curved brackets. We will be using the curved brackets to pass **parameters** later.



**{ }**

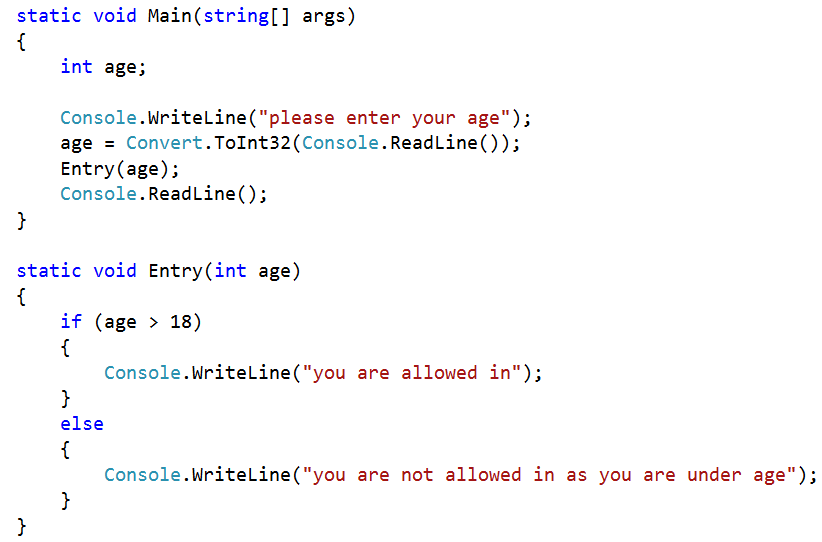
Coding Task: *Program 6.1 – The Secret Word Program*

Copy the code from Program 6.1.txt into a new project. Adjust the program so that there are  
separate procedures for:

* outputting the menu
* making a word guess
* changing the secret word
* quitting the program

## Calling procedures with parameters

Sometimes the procedures you call will need to have **values passed** with them so that the procedure can **use** those values to carry out some **task**. The program below is used to work out whether or not a person can be allowed entry into an over-18 venue.



In order for the **Entry ()** procedure to work out whether or not a person is allowed entry to the club, it needs to have the **age** of the person **passed** to the procedure as a value. We call this value an **actual parameter or argument**.

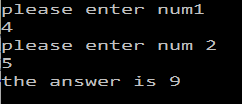
When the entry procedure is written, **the formal parameters or arguments** are declared in the curved brackets next to the procedure identifier. **int** refers to the fact that the **actual parameter** must be an integer data type. The identifier of the formal parameter does not need to be the same name as the actual parameter.



**{ }**

Coding Task: *Program 6.2 – Adding Numbers Together Using Parameters*

Write a program that inputs two numbers and outputs the result of adding them together. The input   
of the two numbers can be done by the main procedure. The **addition** and **output** should be done by a separate procedure called by the main.



## What is a function?

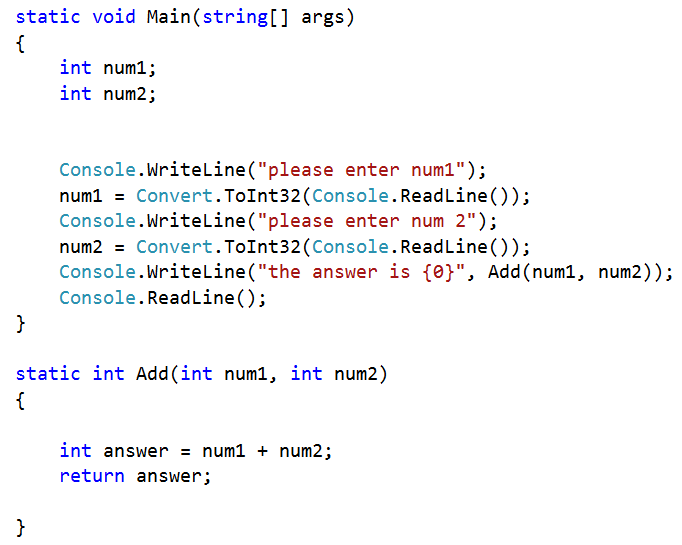
A function is simply a procedure that will **return a value** to the calling procedure. The procedures we have written so far have all been **void** – this means that they have not returned any value to the calling procedure after execution.

Program 6.2 called the procedure **AddNumbers()** to add two numbers together and output the result. The code below shows this program adapted to use a **function** – instead of also outputting the result it returns the **answer** to the calling procedure and then the result is output by the **Main()** procedure instead. A **function** can be used as part of an **assignment statement** as it **returns a value** to the calling procedure.

Every function is a type of procedure or method, but not all procedures are functions.

Functions **always** return a value whereas this is not true of other procedures.

### An example of the syntax used in functions



As the function returns a value, we can no longer use the void keyword; instead, we use the **data type of the return value**.

As this is a function it must have a **return** statement.   
This sends the value back to the calling procedure.

The function call with parameters passed.



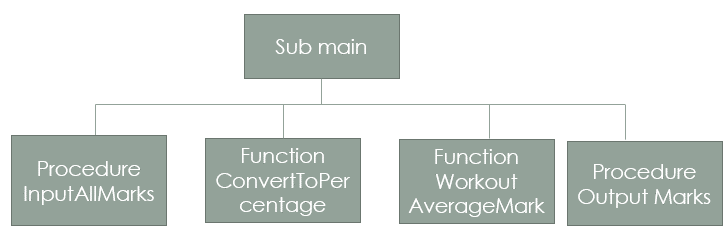
**{ }**

Coding Task: *Program 6.3 Calculating Percentages Using Functions*

A teacher has set her students a programming exam paper. She has five students in her class. The marks for her students out of 60 are (22, 31, 44, 56, 22). These marks must be input into the program.

Your program needs to work out and display for this paper:

* the percentage value for each student
* the average percentage value for the class (average number)

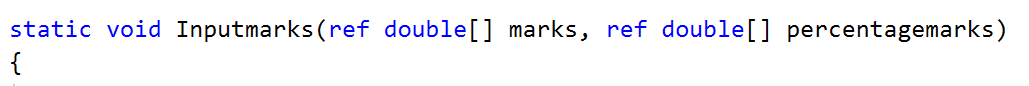


## The ref keyword

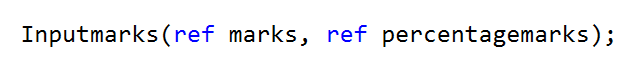
By default, parameters are passed to a procedure or function **‘By Value’**. This means that new memory locations are created for the parameters and, therefore, the value of the original argument is not changed. By contrast, if we use the keyword ‘**ref**’ in front of a parameter variable, new memory locations are not created and, therefore, we are changing the original argument value.

* **By Value** – a value parameter is a piece of data used by the procedure it is passed to. Any changes made to it inside the called procedure are not passed back again.
* **By Ref** – this is used when a piece of data needs to be passed to a procedure, changes made to the original value and the changed value passed back to the calling procedure.

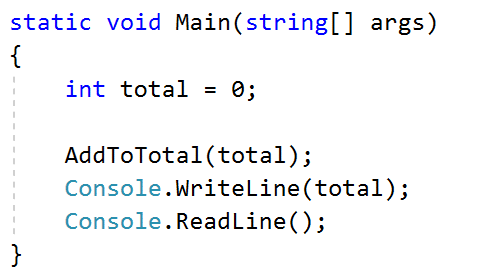
### How to use the ref keyword



The **ref** keyword is placed in front of the formal parameter/argument. This means that changes can be made to the original array (in its original memory location). It is also placed in front of the actual parameter/argument in the procedure call.



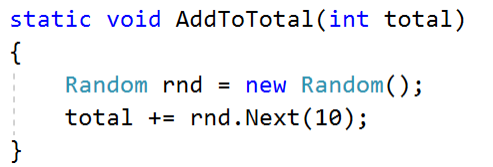
### An example of where the ref keyword would be required



In this program the **Main()** procedure declares a variable called **total** and initialises it to 0.

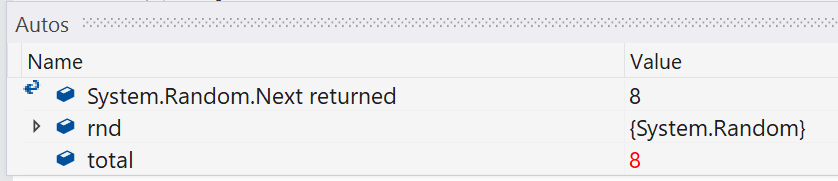
This value is then passed to **AddToTotal()** as a parameter. **AddToTotal()** produces a random number up to 10 and adds it to **total**.

When **AddToTotal()** finishes executing, the **Main()** procedure will output **total** to the console window.



### When the program is run

This program outputs 0 to the console window even though an integer of 8 was generated as a random number to be added to **total**. The image below shows the value of total after execution of **AddToTotal()**.



Try building this in a console application and using the **Step Into** tool (debug menu) to see what happens.

Then try adding the keyword **ref** to the procedure **AddToMarks(ref int total)** and to the procedure call **AddToMarks(ref total)**. This time the new value generated by **AddToMarks()** will be output to the console window.

## Chapter 6 – Consolidation Tasks

Program 6.4

The following algorithm describes the process by which an 8-bit binary number is converted into its denary equivalent. The top-down design diagram shows how the algorithm could be split into separate tasks.

Program the algorithm in C# by selecting appropriate structures and statements and writing separate procedures and functions to represent the different tasks.

|  |
| --- |
| **Algorithm to convert an 8-bit binary number to denary** |
| Answer = 0 |
| Column = 128 |
| OUTPUT “Please enter the binary number to be converted” |
| Binarynumber = USERINPUT |
| FOR I = 0 To Binarynumber.length - 1 |
| Bitvalue = 0 |
| IF binarynumber(i) == “1” THEN |
| bitvalue += column |
| ELSE |
| Bitvalue = bitvalue |
| END IF |
| Column /= 2 |
| Answer += bitvalue |
| END FOR |
| OUTPUT Answer |

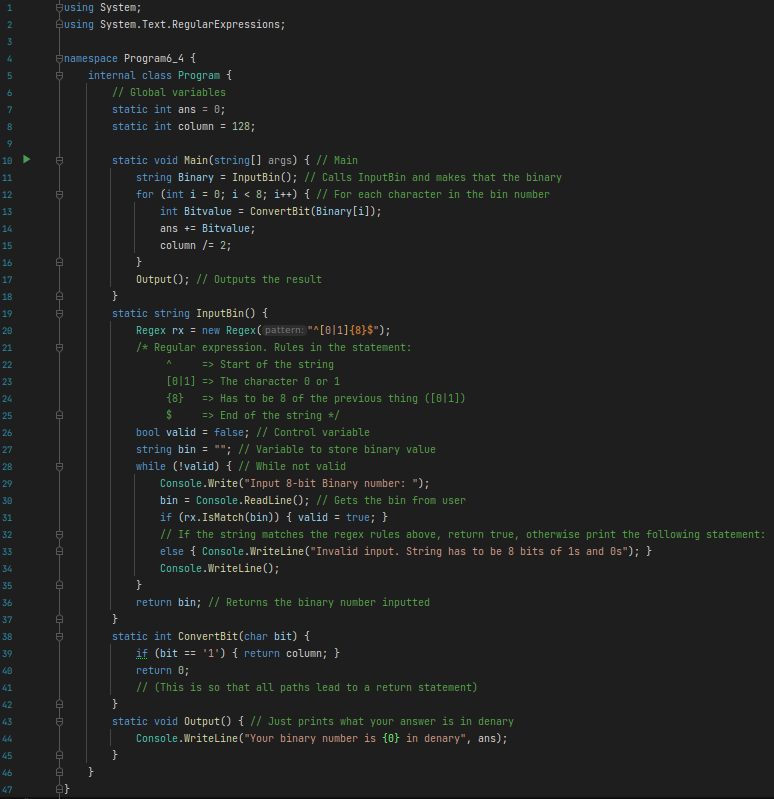
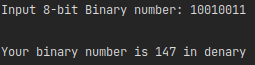
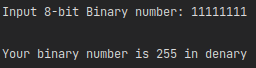
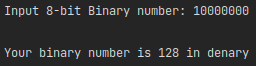
This is the suggested task breakdown for the converter program. **ConvertBit()** should be called for every bit in the binary number.

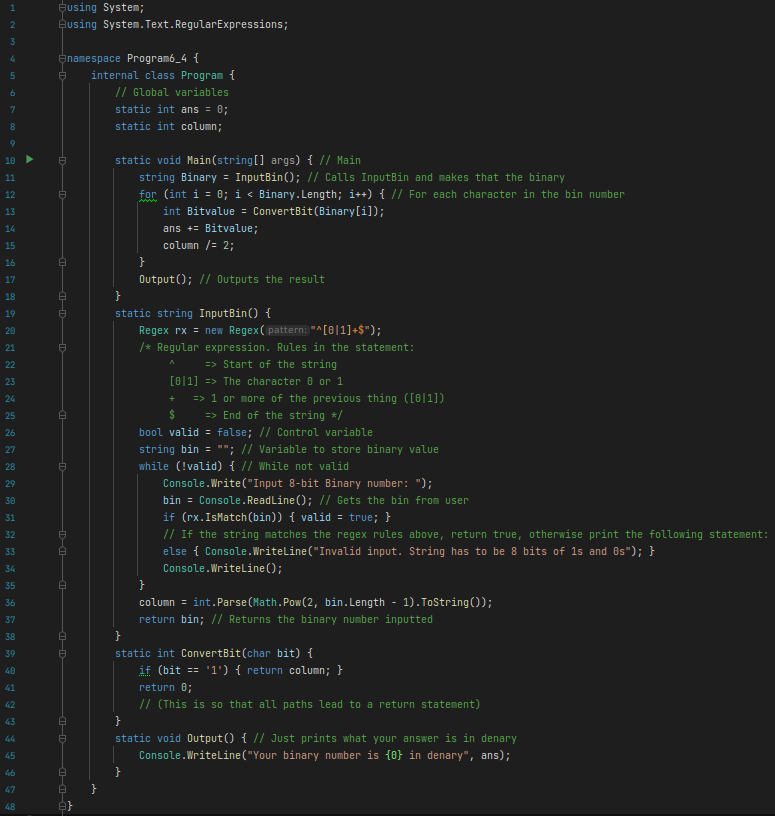
**Evidence required:**

1. **Provide your code listing** for the above program.
2. **Test your program** with the following binary numbers:
   1. 10000000 **128**
   2. 11111111 **255**
   3. 10010011 **147**
3. **Adapt the program** so that it would work for any length of binary number.
   1. **Provide your adapted code** for the above.
   2. **Test** with the following binary numbers:
      1. 0011111110 **254**
      2. 1111 **15**
      3. 001101 **13**

Consolidation Tasks:

1.

2.

3. a)b)