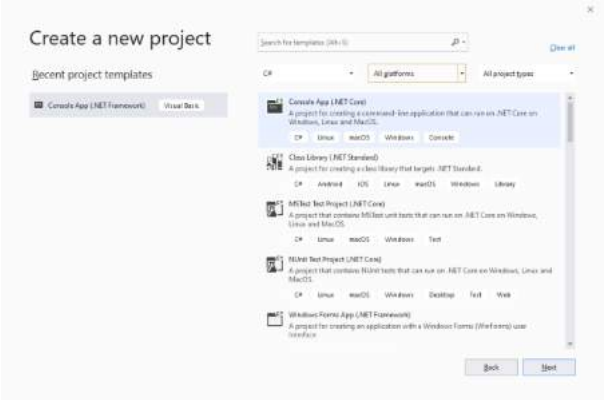
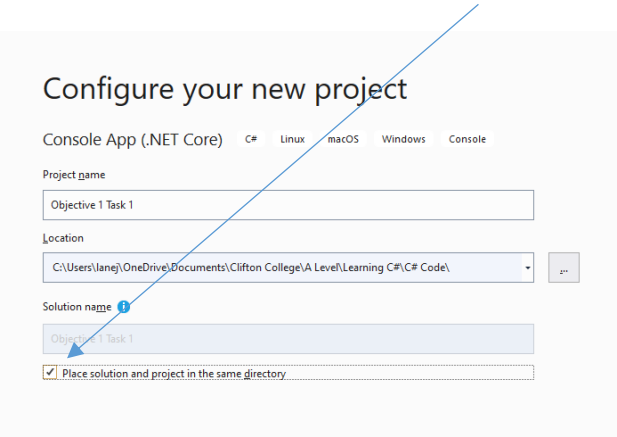
Getting started in console C#

1. Download and install Visual Studio Community Edition. It's free but you will need a free Microsoft account if you haven't already got one to register it. Instructions are linked here. For Step 4, the workload you should install is **.NET desktop development**. The optional steps are optional! If you want to add in other components such as Unity, you can later at any time.
2. Open Visual Studio. This guide is written for the Community 2019 version.
3. Choose Create New Project…
4. Choose C#… Console App (.Net Core)…

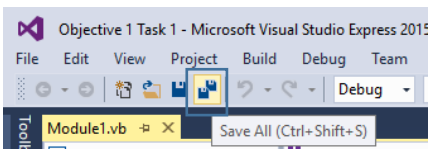


5. Make sure you name your program appropriately, e.g. "Objective 1 Task 1" and choose an appropriate place to save the project. Note, C# programs consist of many individual files grouped into a solution and therefore you should enable 'Place solution and project in the same directory'.



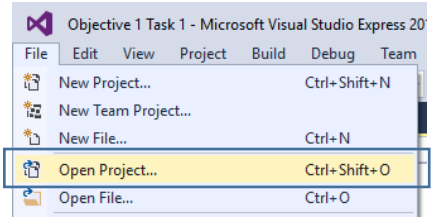
6. To run your program, press F5.

7. When you save your work, make sure you choose Save All



8. When you want to open your work again, choose Open Project…, **not** Open File. Find the .sln file.

9. When working on challenges, remember to put `Console.Readline()` at the end of your program, otherwise the code will run, and end before you can see the output!



Your first program:

Programming languages have a wide variety of tools available. Having access to all of them at one would slow things down so we only import what is needed. 'System' contains the standard set of tools and is the default toolkit. To import a set of tools, we use the 'using' key word.

Semi-colons are used at the end of each program statement

This is the procedure that will run when we run the application, an **entry point** into the application, a procedure or **method** called 'Main':

- Main is in every program and is the default starting procedure.
- 'Static' means we don't need an instance of the class 'Program' to run the code. This will be clearer when we look at Object Oriented Programming later.
- 'Void' means that the procedure does not return any data back to any calling program; (string[] args) is the data the procedure needs to run.

Line numbers

```
1     using System;
2
3     namespace Objective_1_Task_1
4     {
5         0 references
          class Program
6         {
7             0 references
              static void Main(string[] args)
8             {
9                 Console.WriteLine("Hello World!");
10            }
          }
      }
```

A namespace is simply an organisational container used to keep related code together.

A 'class' is another type of organisational unit that describes what the code within it can do and what properties it has. The default class is 'Program' and within this class we write code that runs within this program. We can call (refer to and run methods in) other classes and we will look at this in Object Oriented Programming.

Curly brackets (aka '**braces**') are used to separate different blocks of code

TYPE YOUR CODE HERE

This is our actual code instruction, to write a short line of code to the console. Note the semi-colon.

Objective 1:
Understand how to output text strings

In this objective you learn how to output text to the screen.

Tasks

1. Try entering the following commands and see what happens:

```
Console.WriteLine("Hello World");
```

If the black console window closes straight away then add `Console.ReadLine()` afterwards:

```
Console.WriteLine("Hello World");
Console.ReadLine();
```

2. Try entering the text without the speech marks and see what happens:

```
Console.WriteLine(Hello World);
```

Note the difference. One works and one doesn't! If you want to output text to the screen it must be included in speech marks. Text is called a string in programming and must be enclosed in double quotes.

3. Try entering the following commands and see what happens:

```
Console.WriteLine("Hello" + " this is my first program.");
```

Note how a plus symbol (+) joins strings together. This is called concatenation.

4. Try this sequence of instructions:

```
//Start of message
Console.WriteLine("Objective 1 Tasks:");
Console.WriteLine("------------------");
Console.WriteLine("Hello" + " this is my first program.");
//Blank line
Console.WriteLine();
//End of message
Console.WriteLine("I am learning to code...");
Console.WriteLine("...and it is fun");
//Wait for user to exit (if necessary)
Console.ReadLine();
```

Note the double forward slash symbol (//) enables you to insert a comment in the code. The computer ignores the comments, but they help to introduce the program and allow you to leave notes in the code to make it easier to understand later.

5. Change the program so it outputs this instead:

```
Computers only do exactly as they are told…
…so you need the code to be correct!

If you make any mistake with the commands, it won't work
```

Objective 1: Key learning points

Understand how to output text strings

- Text is known as a **string** in programming and must be enclosed in double quotes.
- Strings can be joined together using a plus symbol. This is known as **concatenation**.
- When a program is run it is called **executing** the program.
- A set of instructions that execute one after another is known as a **sequence**.
- Comments can be inserted into the code using a double // symbol. These are helpful to make notes within the program so that the code can be understood again at a later date or by another programmer.

Objective 1: Key words

Console.Writeline()

Example code: `Console.Writeline(x);`

Purpose: to output x to the screen followed by a carriage return.

Console.Write()

Example code: `Console.Write(x);`

Purpose: to output x to the screen without a carriage return.

Comments

Example code: `//Wait for user to exit`

`//` works for single lines but for multi-line comments you can use this format:

```
/* This is the main program that is run first in all applications.
* Each learning objective is called from different classes
* Note that this works for multi-line comments
*/
```

Objective 2:

Understand how to input strings and numbers into variables

In this objective you learn how to get input from the keyboard to use in your program.

Tasks

1. Try entering the following commands and see what happens:

```
//Declare variables
//The type of data comes first, then the variable name or identifier
string name_entered;
//Inputting strings
Console.WriteLine("Hello");
Console.WriteLine("What is your name?");
name_entered = Console.ReadLine();
Console.WriteLine("Thank you " + name_entered);
//note the variable concatenated on the end
```

2. Try entering the following commands and see what happens:

```
int year;
Console.WriteLine("What year is it please? ");
year = Console.ReadLine();
Console.WriteLine("Ah, it is " + year + " thank you.");
```

This should produce an error: `year = Console.ReadLine();` because whatever you type in using Console.ReadLine() has a data type of string. The variable 'year' has been declared as an integer (a whole number) so we must convert the input into an integer. Change that line of code to:

```
year = Convert.ToInt32(Console.ReadLine());
```

Where `Convert.ToInt32` converts the input into a 32-bit integer.

Now try using "composite formatting" to output the final line. This works as a shorthand especially if you have lots of variable names to include in your output:

```
// Composite formatting method of output:
Console.WriteLine("Ah, it is {0} thank you.", year);
```

The value(s) in braces {} are replaced by the variables listed at the end of the statement in the order they appear.

3. Change the program so it asks you for your name and your age, outputting for example:

```
Thank you Dave. You have registered an age of 15.
```

Objective 2: Key learning points

How to input strings and numbers into variables

- Data is input by a user into a **variable**.
- Memory is reserved for variables when you **declare** the variable.
- Variables have a data type: string, integer or decimal as examples, indicating how much memory they will use and the type of data they will store.
- Variables must be declared before they can be used.
- Variable declaration takes the form of: `data_type identifier;`

Objective 2: Key words

Variable

A named space in memory reserved to store some data (that can change) to be used in the program.

Example code: `type x;`

Purpose: to declare a variable to be used later in the program.

x is the Identifier of the variable. `type` is the data type for the variable. Examples of `type` include:

- `string` (text)
- `char` (single unicode character)
- `int` (whole number)
- `float` (number with decimal places)
- `double` (decimal number with a greater range & accuracy but takes up more memory)
- `decimal` (decimal number with even greater accuracy but takes up more memory, used for money in financial transactions to ensure accuracy)
- `bool` (Boolean value: either true or false)
- `DateTime` (date and/or time)

Console.Readline()

Example code: `x = Console.Readline();`

Purpose:

- to store text input at the keyboard into a variable, x which can be used later in the program without inputting again.
- Halt the command line interface waiting for an input eg to exit

Console.Writeline() with variables

Example 1 using concatenation to include variables:

```
Console.WriteLine("Thank you " + name + ". You have registered an age of " + age + ".");
```

Example 2 using composite formatting to include variables:

```
Console.WriteLine("Thank you {0}. You have registered an age of {1}.", name, age);
```

The values in braces {} are replaced by the variables listed at the end of the statement in the order they appear. This also allows a wide variety of formatting options. Further information is linked here.
Experiment!

Objective 3:

Understand string manipulation functions

In this objective you learn how to extract and change data that has been input.

Tasks

1. Try entering the following commands and see what happens:

```
// .ToUpper() changes a string to upper case
string forename, forename_uppercase;
Console.Write("Enter your surname: ");
forename = Console.ReadLine();
forename_uppercase = forename.ToUpper();
Console.WriteLine("Your name in capital letters is: " + forename_uppercase);
```

2. Change the program to ask for an email address, outputting the string in lowercase using `ToLower()` instead of `ToUpper()`.

3. Try entering the following commands and see what happens:

```
//.Length returns the number of characters in a string
string surname;
int length_name;
Console.Write("Enter your surname: ");
surname = Console.ReadLine();
length_name = surname.Length;
Console.WriteLine("There are " + length_name + " letters in your name.");
```

4. Try entering the following commands and see what happens:

```
/* task 4
* Substring is used to slice up strings
* The position of the first character in a string is always 0
* Known as 'zero-based strings' */
string sentence, characters;
sentence = "I saw a wolf in the forest. A lonely wolf.";
// Note below the 0 could be omitted; 0 is the default
characters = sentence.Substring(0, 5);
// alternative: characters = sentence.Substring(5);
Console.WriteLine("First 5 characters: " + characters);
```

5. Try entering the following commands and see what happens:

```
//task 5: characters in the middle
string sentence, characters;
sentence = "I saw a wolf in the forest. A lonely wolf.";
characters = sentence.Substring(21, 6);
Console.WriteLine(characters);
```

6. Change the program to output the last 12 characters in the sentence using the Length function of the string.

7. Try entering the following commands and see what happens:

```
//task 7:
//.IndexOf returns the location of one string inside another
string sentence, word;
int position;
sentence = "I saw a wolf in the forest. A lonely wolf.";
Console.WriteLine(sentence);
Console.WriteLine("Enter the word to find: ");
word = Console.ReadLine();
position = sentence.IndexOf(word);
Console.WriteLine("The word " + word + " is at character " + position);
// Note alternative composite formatting output method
Console.WriteLine("The word {0} is at character {1}", word, position);
```

Note:

- the first character in a string is at position 0, so it starts counting from 0.
- If the searched for item isn't in the string, then it will return -1.
- Using `LastIndexOf()` method searches backwards from right-to-left

Objective 3: Key learning points

String manipulation functions

- Strings can be manipulated using the built in `Substring` function to extract characters from the left, right or middle of a string.
- You can find if one string exists inside another string using `IndexOf`.
- A built in function takes data to use in parenthesis (brackets), called a **parameter** and returns a result.

Note, it is possible for functions to return their value to another command rather than a variable. For example: `Console.WriteLine("Hello World".IndexOf("World"))` works because the resultant value from `IndexOf` becomes the parameter for Console.Writeline.

It is common in programming to use as few variables as necessary in order to conserve memory. This makes the program more efficient, but often more difficult to understand.

Objective 3: Key words

ToUpper()

Example code: `x = y.ToUpper()`

Purpose: To turn a string into uppercase. x is the name of the variable to return the result to. y is the parameter.

ToLower()

Example code: `x = y.ToLower()`

Purpose: To turn a string into lowercase.

Length

Example code: `x = y.Length`

Purpose: To return the number of characters in a string. x becomes the number of characters in y.

Substring()

Example code: `x = y.Substring(z)`

Purpose: To return characters to the left of a string.

x becomes the characters. y is the string to extract characters from. z is the number of characters to extract from the left.

Substring()

Example code: `x = w.Substring(y,z)`

Purpose: To return characters from the middle of a string.

x becomes the middle characters of w, starting from position y, extracting z number of characters.

IndexOf ()

Example code: `x = y.IndexOf(z)`

Purpose: To find the position of substring z in y.

X becomes the position in the string y where z can be found. E.g. x = "Hello World".IndexOf("World") returns 7 as the letter W is the seventh character in the string.

Objective 4:

Understand how to use selection statements

In this objective you learn how to make different parts of your program run depending on the value of variables.

Tasks

1. Try entering the following commands and see what happens:

Note no semi-colon after this statement

Note the use of braces { } to separate each code block

```
//Using selection statements to check variables
//Declare a number variable
int number;
//Ask user for the number
Console.WriteLine("Enter a number 1-3: ");
number = Convert.ToInt32(Console.ReadLine());
//Check the value of the number
if (number == 1)
{
    Console.WriteLine("You entered one");
}
else if (number == 2)
{
    Console.WriteLine("You entered two");
}
else if (number == 3)
{
    Console.WriteLine("You entered three");
}
```

2. Try entering the following commands and run the program three times with the test data: 6, 51 and 70.

```
//Works out whether a candidate passed or failed
int score;
Console.WriteLine("Enter the test score 1-100: ");
score = Convert.ToInt32(Console.ReadLine());
//Branch depending on the value of the variable
if (score > 40)
{
    Console.WriteLine("You passed");
}
else
{
    Console.WriteLine("You failed");
}
```

Note how the program only executes instructions if the condition is true i.e. the score is more than 40, and how it is possible to use an else statement to run an alternative set of commands if the condition is false.

3. Change the program so that you must score over 50 to pass.

4. Try entering the following commands and run the program three times with the test data: 6 & 4, 51 & 51.

```
//Returns whether two numbers are the same
int num1, num2;
Console.Write("Enter the first number: ");
num1 = Convert.ToInt32(Console.ReadLine());
Console.Write("Enter the second number: ");
num2 = Convert.ToInt32(Console.ReadLine());

//Branch depending on the value of the variable
if (num1 != num2)
{
    Console.WriteLine("The numbers are different");
}
else
{
    Console.WriteLine("The numbers are the same");
}
```

Note: != means 'does not equal'.

5. Try entering the following commands and run the program three times with the test data: 1, 3, 5.

```
//Using logic operators
int choice;
//Ask user for the number
Console.Write("Enter a number 1-3: ");
choice = Convert.ToInt32(Console.ReadLine());
//Check the value of the number
if (choice > 0 && choice < 3)
{
    Console.WriteLine("Valid input");
}
else
{
    Console.WriteLine("Invalid input");
}
```

Note how '&&' (AND) can be used to check if both conditions are true. You can also use '||' (OR) if only one condition needs to be true and '!' (NOT) as an alternative to does not equal.

6. Try entering the following commands and see what happens:

```
//Using switch selection
//More efficient with 1 variable and several possibilities
string choice;
//Ask user for the number
Console.WriteLine("1. Add numbers\n" +
    "2. Subtract numbers\n" +
    "3. Quit\n" +
    "Enter your choice: ");
choice = Console.ReadLine();
//Multiple branches depending on selection
switch (choice)
{
    case "1":
        Console.WriteLine("Add numbers chosen");
        break;
    case "2":
        Console.WriteLine("Subtract numbers chosen");
        break;
    case "3":
        Console.WriteLine("Quit chosen");
        break;
    default:
        Console.WriteLine("Invalid option chosen");
        break;
}
```

Notes:

- It is possible to use a `switch… case…` structure to have multiple branches rather than just true or false.
- In C#, each `case` statement must be followed by a `break` to bring it out of the `switch` code block.
- The `default` option is optional but good practice. It is what gets run when the variable entered does not meet any of the other conditions

7. Comparing strings. Try the following code and see what happens when you enter 'hello' and 'hello', 'hello' and 'Hello':

```
string str1, str2;
Console.Write("Enter first string: ");
str1 = Console.ReadLine();
Console.Write("Enter second string: ");
str2 = Console.ReadLine();
// use standard == operator
if(str1 == str2)
{
    Console.WriteLine("Strings '{0}' and '{1}' match.", str1, str2);
}
else
{
    Console.WriteLine("Strings '{0}' and '{1}' don't match.", str1, str2);
}
```

8. Now try the same thing but replace the '==' in the if statement with '>'. What happens?
9. For putting strings such as names into an order such as alphabetical, we do need to be able to compare them properly. Try the following code:

```
string str1, str2;
//int chr1, chr2;
Console.Write("Enter first string: ");
str1 = Console.ReadLine();
Console.Write("Enter second string: ");
str2 = Console.ReadLine();
// use String.Compare
if (String.Compare(str1, str2, StringComparison.Ordinal) == 0)
{
    Console.WriteLine("Strings '{0}' and '{1}' match.", str1, str2);
}
else if (String.Compare(str1, str2, StringComparison.Ordinal) < 0)
{
    Console.WriteLine("String '{0}' is less than string '{1}'.", str1, str2);
}
else
{
    Console.WriteLine("String '{0}' is greater than string '{1}'.", str1, str2);
}
```

```
String.Compare(str1, str2, StringComparison.Ordinal
```

`String.Compare` compares two strings, in this case str1 and str2, and returns:

- 1 if the first string is highest
- -1 if the second string is highest
- 0 if they match

There are many ways to compare strings depending on the local alphabet being used (known as 'culture') so we must state what type of comparison we are going to use.

Here we have used `StringComparison.Ordinal`, which compares the ASCII/Unicode integer character code for each character in the string and note that, counter-intuitively, capital letters have a lower code value than lower-case letters. If we used `StringComparison.OrdinalIgnoreCase`, it would ignore the case of the letters.

Objective 4: Key learning points

How to use selection statements

- Checking the value of a variable and executing instructions depending on the outcome of the check is known as a **selection**.
- The instructions that are executed as a result of a selection is known as a **program branch**.
- The logical checking of a variable against another value is known as a **condition**.
- Switch/case commands can be used instead of multiple if statements.
- Code for each program branch should be indented.
- It is good practice to comment a selection to explain its purpose.
- One 'If' can be placed inside another 'If', this is known as **nesting**.

Objective 4: Key words

If… elseif … else

Example code:

```
if ((x>0 && x<5) || x==10)
{
            …;
}
else if (x > 10)
{
            …;
}
else
{
            …;
}
```

x is the variable being checked. Multiple variables can be checked in one condition. Use brackets to express order of priority. In the example code, x must be between 0 and 5 or equal to 10 for the condition to be true. The else if checks if x is bigger than 10. Else is optional and is used to branch if the condition is false. If conditions can test more than one variable.

Logical operators that can be used in conditions include:

==	equals	<	less than	&&	both conditions must be true (AND)
!=	does not equal	<=	less than or equal to	||	one condition must be true (OR)
		>	greater than	!	condition is not true (NOT)
		>=	greater than or equal to		

Switch… case… break… default

Example code:

```
switch (x)
{
	case 3:
		…;
		break;
	case 4:
		…;
		break;
	case 10:
		…;
		break;
	case 20:
		…;
		break;
	default:
		…;
		break;
}
```

x is the integer variable being checked. Switch… Case is used when there is one variable to be checked and as an alternative to multiple if statements.

String.Compare(string1, string2, comparison_method)

Example code:

```
if (String.Compare(str1, str2, StringComparison.Ordinal) == 0)
{
	Console.WriteLine("Strings '{0}' and '{1}' match.", str1, str2);
}
```

Objective 5:
Understand how to use arithmetic operations, random numbers and type conversions

In this objective you learn how to perform mathematical calculations using power, modulus and integer division. Random numbers allow for some unpredictability which is useful for games. You also learn how to convert from one data type to another.

Tasks

1. Try entering the following program to see how arithmetic operators work using 12 and 3 then 9 and 4 as your input pairs:

```
// Declare variables
int power_of_result;
int number1, number2;
double division_result;
int integer_division_result;
int modulus_result;

// Get user input
Console.Write("Enter first number: ");
number1 = Convert.ToInt32(Console.ReadLine());
Console.Write("Enter second number: ");
number2 = Convert.ToInt32(Console.ReadLine());

// Make calculations
power_of_result = (int)Math.Pow(number1, number2); //Note shorthand conversion
division_result = number1 / number2;
integer_division_result = number1 / number2;
modulus_result = number1 % number2;

// Output results
Console.WriteLine();
Console.WriteLine("{0} to the power of {1} is {2}", number1, number2, power_of_result);
Console.WriteLine("{0} divided by {1} is {2}", number1, number2, division_result);
Console.WriteLine("{0} integer divided by {1} is {2}", number1, number2, integer_division_result);
Console.WriteLine("{0} divided by {1} has a remainder of {2}", number1, number2, modulus_result);
```

What happened to the division?

2. Now try the following with the same inputs and compare the results.

```
// Declare variables with double data type for decimal values
double number1, number2;
double division_result;

// Convert user input into doubles
Console.Write("Enter first number: ");
number1 = Convert.ToDouble(Console.ReadLine());
Console.Write("Enter second number: ");
number2 = Convert.ToDouble(Console.ReadLine());

division_result = number1 / number2;
Console.WriteLine();
Console.WriteLine("{0} divided by {1} is {2}", number1, number2, division_result);
```

3. Try entering the following commands and see what happens:

```
//Declare variables
Random rnd = new Random(); //Generate a new random number object
int random_number;
//Roll the dice
random_number = rnd.Next(1, 7);
Console.WriteLine("You rolled a " + random_number);
```

4. Change the program so that it outputs a 10 sided dice.

5. Change the program so the user can choose to roll a 4, 6 or 12 sided dice.

Objective 5: Key learning points

How to use arithmetic operations and random numbers

- / performs division. If the operand data types are integers, it does integer division, giving an integer answer. If the operands are decimal data types (eg double), then it performs regular division.
- **Modulus** is the remainder of a division. E.g. 8 divided by 5 is one with 3 left over (remainder)
- Commands can be put inside commands. E.g. Convert.ToInt32(Console.ReadLine())is two commands, one inside the other. This example is converting an input into an integer.
- A variable is stored with a specific data type that determines how much memory it uses and how it is handled in the program. Typical data types include: integer, real/float (double/decimal), character, string, date and boolean.
- Some data types have identifying characters. E.g. a string is enclosed in quotes.
- The integer 7, is therefore not the same as the character "7". They have different binary values.
- It is possible to change a variable's data type using conversion functions (Convert).

Mathematical operators that can be used in conditions include:

+	addition	Math.Pow(num, exponent)	*num* to the power *exponent*
-	subtraction	/	integer division occurs with integer values
*	multiplication	%	modulus (remainder after division)
/	division		

Objective 5: Key words

Math. ()

The Math class, used above, contains lots of different mathematical methods for operations such as rounding, PI and trigonometry. For further details, see here.

%

Example code: `x = y % z`

x becomes the remainder of y divided by z. This is useful to determine if one number is exactly divisible by another. For example with x = 2000 % 4, x would be 0 because 2000 is exactly divisible by 4.

Random()

Example code: `Random r = new Random();`

This creates a new random number object called 'r' which can be used to generate random numbers from. Computers are not able to generate truly random numbers. Instead they make a calculation using "seed" numbers and return numbers from a sequence. C# in .NET Core provides a seed value automatically (for more details see here) but you could specify your own.

Next()

Example code: `x = r.Next(y,z)`

x becomes a random number between y (INCLUSIVE) and z (EXCLUSIVE) generated from the random number object 'r'. There are different forms:

- `r.Next()` produces a number from 0 (inclusive) upwards, depending on data type
- `r.Next(y)` produces a number from 0 (inclusive) up to, but NOT including, y
- `r.Next(y,z)` produces a number from y (inclusive) up to, but NOT including, z

Convert.ToInt32()

Example code: `x = Convert.ToInt32(y)`

x becomes the integer of y, or y rounded to the nearest whole number towards 0.

Convert.ToDouble()

Example code: `x = Convert.ToDouble(y)`

x becomes the decimal value of y

Convert.ToString()

Example code: `x = Convert.ToString(y)`

x becomes a string of y

Converting data types (called **casting**) can become complicated because variables of a particular type have a specific amount of memory reserved for them. In some cases, if you are converting to a type that uses more memory, then it won't lose data and can work easily. This is called an **implicit** casting. If the conversion may result in some data being lost, then the conversion must be made **explicitly**. For example:

`int num1 = 5;`
`double num2 = num1;`
This will work because no accuracy is lost, double takes up more memory than decimal.
	`double num3 = 5.0;`
	`int num4 = num3;`
	This won't work because accuracy is lost, double takes up more memory than int. We have two options to fix this:
	`num4 = Convert.ToInt32(num3)`
	Or
	`num4 = (int)num3`

Note that the shorter conversion, `num4 = (int)num3`, can only be used when converting certain numeric data types, typically from decimal types to integer types. Further information can be found here.

Objective 6:

Understand counter controlled iterations

In this objective you learn how to get code to repeat itself a given number of times without having to enter lots of repeating statements.

Tasks

1. Try entering the following commands and see what happens:

```
for(int counter = 1; counter <= 5; counter++)
{
    Console.WriteLine("This is how you get code to repeat");
}
```

2. Try changing the number 5 and see what happens. Also try changing the number 1 and see what happens.

3. Modify the code as shown below. Run the program and see what happens. Note how the value of the counter variable changes.

```
for (int counter = 7; counter <= 12; counter++)
{
    Console.WriteLine("The counter is {0}", counter);
}
```

4. Now change the counter parameter to say `counter-=2` instead of `counter++` and see what happens. Now fix the code so that it counts in twos and stops at 12.

5. Enter the following commands and see what happens:

```
string word = "Hello";
for (int counter = 0; counter < word.Length; counter++)
{
    Console.WriteLine("Letter {0} is a '{1}'", counter, word.Substring(counter, 1));
}
```

Note how we can now loop through all the letters in a word by combining an iteration with string manipulation commands. Also note how commands inside an iteration are indented to make the block of code easier to read.

6. This returns the ASCII/Unicode value **for each** letter (remember Objective 4 task 9?) by casting the character 'letter' into an integer. Try the code out for words such as Hello, heLlo, AaBbCc:

```
Console.Write("Please enter a word: ");
string word = Console.ReadLine();
foreach (char letter in word)
{
    Console.WriteLine("Letter {0} has an ASCII/Unicode value of {1}", letter, (int)letter);
}
```

Objective 6: Key learning points

Counter controlled iterations

- An **iteration** is a section of code that is repeated.
- Iterations are also known as **loops**.
- Definite, or count-controlled iterations are used when you want to iterate a known number of times.
- Counter controlled iterations use a variable (the counter) which can be used within the code to know how many times the code has been run.
- Code is **indented** within an iteration to make it easier to read.
- It would be good practice to comment code before an iteration to explain what is happening.
- **Incrementing** a variable increases its value by one.
- **Decrementing** a variable decreases its value by one.
- The incrementing or decrementing of the counter variable happens automatically.
- One 'For' loop can be put inside another 'For' loop. This is known as **nesting**.

Objective 6: Key words

For()

Initialising the starting value of the counter variable as an integer

The stopping condition for the loop

The 'step': how the loop counter increments; -= means count backwards, also can count in 2s or anything by (eg) counter += 2

Example code:

```
for(int counter = 1; counter <= 5; counter++)
{
            Console.WriteLine(counter);
}
```

Executes code within the 'for' structure a known number of times. `counter` is the variable that is counting how many times the code has iterated. 1 is the starting value.

The **stopping condition** controls how many times the block repeats; in this case, the code will repeat while the `counter` is less than or equal to 5. Note that this is a Boolean statement, either TRUE or FALSE.

Step determines by how much the `counter` increments. The standard is a step of +1, abbreviated to '++'. To count backwards, a negative value can be used for step e.g. -= 1.

Foreach()

Example code:

```
string word = "Hello";
foreach (char letter in word)
{
      Console.WriteLine("Letter {0} has a Unicode value of {1}", letter, (int)letter);
}
```

`foreach` can be used for sequential collections like strings, arrays and lists (see later). Instead of using a counter, `foreach` looks at each item in the sequence one-by-one in order. A string is a sequence of characters so the `char` data type is used to look at each letter in the `string word` in turn.

Objective 7:

Understand condition controlled iterations

In this objective you learn how to get code to repeat continually until a condition is met.

Tasks

1. Try entering the following program and see what happens by entering a few numbers and ending by entering zero:

```
//Program to add up numbers until 0 is entered
//Initialise variables
double total = 0;
double number_entered = 1;
//Enter numbers to add
while (number_entered > 0)
{
    Console.Write("Enter a number: ");
    number_entered = Convert.ToDouble(Console.ReadLine());
    //Add number to total
    total += number_entered;
}
//Output result
Console.WriteLine("The sum of the numbers is: " + total);
```

2. Compare the program above to this code written slightly differently:

```
//Program to add up numbers until 0 is entered
//Initialise variables
double total = 0;
double number_entered;
//Enter numbers to add
do
{
    Console.Write("Enter a number: ");
    number_entered = Convert.ToDouble(Console.ReadLine());
    //Add number to total
    total += number_entered;
}
while (number_entered > 0);   //condition is at the end
//Output result
Console.WriteLine("The sum of the numbers is: " + total);
```

No need to initialise this variable

Note the semi-colon

Note that the `while` statement can be placed either at the beginning of the iteration (as in program 1) or at the end of the program (as in program 2). Although these iterations both use the same commands, they are not the same.

When the `while(number_entered > 0)` condition is **placed at the start**, the code inside the iteration **may not execute at all** because the value of number_entered is checked first.

When the `while(number_entered > 0)` condition is **placed at the end** and used with the `Do` statement at the start, the code inside the iteration **must execute at least once** because the value of number_entered is not checked until the end.

Objective 7: Key learning points

Condition controlled iterations

- When it is not known in advance how many times code needs to be repeated, a condition controlled loop is used.
- Conditions can be set at the end of the iteration so code executes at least once, or at the beginning of the iteration where code inside the iteration may not execute at all.
- Condition controlled iterations are useful when you want to validate data entry by only continuing to the next statements once the data input is correct.
- There are many examples of condition controlled iterations, `while` and `do… while` are just two examples.
- Code should always be indented inside iteration statements.
- The conditions return a Boolean value; they are TRUE or FALSE.

Objective 7: Key words

While ()…

Example code:

```
while (counter < 3)
{
            …
}
```

Checks the condition and if true executes the code in the braces, returning to check again afterwards.

Do… While()

Example code:

```
do
{
            …
}
while (counter < 3);
```

Executes the code in the braces after the `do` statement at least once before checking the condition. If the condition is true, the code after to `do` statement executes again.

Objective 8:

Understand subroutines, procedures and functions

In this objective you learn how to divide your program into more manageable sections to make it easier to read and reduce the need for entering repeating code.

Tasks

1. Try entering the following program and see what happens:

```
//Global variables accessed by all subroutines
static int MaxNoOfStars, NoOfStars, NoOfSpaces;
0 references
static void Main(string[] args)
{
    InitialiseNoOfSpacesAndStars(); //Main program starts here
    do
    {
        OutputLeadingSpaces();
        OutputLineOfStars();
        AdjustNoOfSpacesAndStars();
    }
    while (NoOfStars <= MaxNoOfStars);
}

//Set the size of the pyramid of stars
1 reference
static void InitialiseNoOfSpacesAndStars()
{
    Console.Write("How many stars at the base (1-40)? ");
    MaxNoOfStars = Convert.ToInt32(Console.ReadLine());
    NoOfSpaces = MaxNoOfStars / 2;  //Calculate spaces needed from tip
    NoOfStars = 1;                  //Set tip of pyramid to one star
}

//Outputs spaces before stars
1 reference
static void OutputLeadingSpaces()
{
    for (int count = 1; count <= NoOfSpaces; count++)
    {  Console.Write(" "); }
}

//Outputs stars
1 reference
static void OutputLineOfStars()
{
    for (int count = 1; count <= NoOfStars; count++)
    {  Console.Write("*"); }
    Console.WriteLine();            //Move to next line
}

//Adjusts number of stars & spaces after output
1 reference
static void AdjustNoOfSpacesAndStars()
{
    NoOfSpaces = NoOfSpaces - 1;
    NoOfStars = NoOfStars + 2;
}
```

Note this goes ABOVE the Main method where you normally type the code

Main method

Note the for loop has been condensed just to fit it on the page

2. Try entering the following program and see what happens:

```
static void Main(string[] args)
{
    // task 2a: 'Program to output a set of random numbers using parameters
    int number, max;
    Console.Write("How many numbers do you want to output? ");
    number = Convert.ToInt32(Console.ReadLine());
    Console.Write("What is the maximum number? ");
    max = Convert.ToInt32(Console.ReadLine());
    outputrandoms(number, max);  // data sent into the procedure
}

// task 2b: Output random numbers
// note parameters in brackets after the method name
1 reference
static void outputrandoms(int n, int m)
{
    int counter, randomnum;
    Random rnd = new Random();
    m += 1; // because the max value in Next() is EXCLUSIVE
    for (counter = 1; counter <= n; counter++)
    {
        randomnum = rnd.Next(m);
        Console.WriteLine("Number {0} is {1}", counter, randomnum);
    }
}
```

Procedure is called here

There are two ways to share data between subroutines. In the first program, the variables were declared outside of any subroutine. That means they are global variables to the program, and their values can be accessed, shared and changed by any subroutine in the program.

In the second program the variables 'number' and 'max' are declared in subroutine main(). Their values are then passed into the subroutine `Outputrandoms`. The values of the variables become `n` and `m` in the subroutine `Outputrandoms`. The variables `n` and `m` are different variables to `number` and `max`, but the value stored in number and max was passed into `n` and `m`. This is also known as passing by value (as opposed to passing by reference, the variable memory address). In this example, the variables `n` and `m` are lost at the end of the `Outputrandoms` subroutine. That means they are local variables to the subroutine.

3. Try entering this program and see what happens:

Note the word 'void' is now replaced by the data type of the return value.

```
static void Main(string[] args)
{
    // task 3a: How functions can be used
    int num1, num2;
    Console.WriteLine("Enter the first number: ");
    num1 = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Enter the second number: ");
    num2 = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("The floor number is: " + Floor(num1, num2));
}

1 reference
static int Floor(int a, int b)
{
    // task 3b: Returns a positive number from subtraction
    int f;
    f = a - b;
    if (f > 0) { return f; } else { return 0; }
}
```

Function is called here.

4. Change the function so it receives the length of two sides of a right-angled triangle and returns the length of the hypotenuse ($h^2 = o^2+a^2$). Remember to use Math(). What is the data type of the hypotenuse?

Objective 8: Key learning points

Understand subroutines, procedures and functions

- **Subroutines** are **sections of code** to break a longer programs into smaller pieces. Thereby making them easier to read and more manageable for teams of programmers to work together on one program.
- Subroutines are also known as **procedures** or **methods**.
- **Functions** carry out small tasks on data by taking one or more **parameters** and **returning** a result.
- To run a subroutine, you **call** it using its name (known as its **identifier**).
- Subroutines/procedures may also take parameters, but do not return a result.
- It is good practice to comment each subroutine or function to explain its purpose.
- Variables declared outside of all subroutines are known as **global variables**. They are available to all subroutines and functions.
- Global variables are not memory efficient since they hold memory for the entire time a program is running. They should therefore be kept to a minimum.
- Variables declared inside a subroutine or function are known as **local variables**. They lose their value when the routine ends, releasing memory.
- Passing variables between subroutines is known as **parameter passing**.

Objective 8: Key words

static void … ()

Example code:

```
static void max (int x, int y)
{
            …
}
```

Creates a subroutine called `max`. Two parameters are passed into `max`, with local variables x and y holding their values.

`max` would be called with: `x = max(6,9)`
6 would be passed as a parameter to variable x. 9 would be passed as a parameter to variable y.

static *<return_data_type>* ..()

Example code:

```
static double max (int x, int y)
{
            double z;
            …
            return z;
}
```

Creates a function called `max`. Two parameters are passed into `max`, with local variables x and y holding their values.

`max` would be called with: `n = max (6,9)`
6 would be passed as a parameter to variable x. 9 would be passed as a parameter to variable y.
The value returned to n would be z, which in this case is a `double` because the function was declared with `double` to indicate the data type of the return value.

Objective 9:
Understand arrays and lists

In this objective you learn how to store multiple items without having lots of different variables.

Tasks

1. Try entering the following program and see what happens:

```
//Declare array and data set
string[] sentence = new string[5] { "The", "quick", "grey", "fox", "jumps" };
//Output contents of array
Console.WriteLine(sentence[0]);
Console.WriteLine(sentence[1]);
Console.WriteLine(sentence[2]);
Console.WriteLine(sentence[3]);
Console.WriteLine(sentence[4]);
```

`sentence` is an example of an **array**: a group of elements with the same name and data type.

Think of an array as a table:

Identifier:	sentence				
Index:	0	1	2	3	4
Data:	The	quick	grey	fox	jumps

2. Try this modification to the program to allow the user to change the colour of the fox:

```
//Declare array and data set
string[] sentence = new string[5] { "The", "quick", "grey", "fox", "jumps" };
Console.Write("Enter new colour: ");
sentence[2] = Console.ReadLine();
//Output contents of array
Console.WriteLine(sentence[0]);
Console.WriteLine(sentence[1]);
Console.WriteLine(sentence[2]);
Console.WriteLine(sentence[3]);
Console.WriteLine(sentence[4]);
```

Note how it is possible to change an individual element of an array by referring to the index.

3. You don't always want to populate the array with data when it is declared. For example, to declare an array of 100 elements, you would use the alternative syntax:

```
string[] items = new string[100];
```

Note this is the NUMBER of elements in the array; this is the value you would get back if you found the length of the array. This is different to the maximum index value, which is one less, in this case 99.

You can then put data into the array using the syntax:

```
Items[0] = "Alpha";
items[99] = "Omega";
```

Note that array indexes always start at zero. So, one hundred elements is 0-99.

4. The power of arrays often comes by combining them with iterations.
   Enter this code as an alternative to the first program and notice how the amount of code is much reduced when you use a for loop to output each of the elements in the array.

```
//Declare array and data set
string[] sentence = new string[5] { "The", "quick", "grey", "fox", "jumps" };
//Use a loop to cycle through the indexes
for (int counter = 0; counter < sentence.Length; counter++)
{
    Console.WriteLine(sentence[counter]);
}
```

5. Try entering this program and test by entering 8743 when prompted for a product code:

```
//Declare array and set data
string[] product;
product = new string[9]
{"1262", "Cornflakes", "£1.40", "8743", "Weetabix", "£1.20", "9512", "Rice Krispies", "£1.32" };
string product_code;    // Product code to find
int counter;            // Used by iteration
bool found;             // Whether product was found in array

//Ask use for the product code to find
Console.Write("Enter the product to find: ");
product_code = Console.ReadLine();

//Use a loop to cycle through the indexes
counter = 0;
found = false;
do
{
    //Output the product if found
    if (product[counter] == product_code)
    {
        Console.WriteLine(product[counter + 1]);
        Console.WriteLine(product[counter + 2]);
        found = true;
    }
    counter++;
}
while (found == false && counter != 9);

//Output message if product is not found
if (counter == 9) { Console.WriteLine("Product not found"); }
```

- Note how an iteration and Boolean variable can be used to go through all the indexes of an array to find an item using the `do… while()` loop. However, this method requires the counter to incremented in the code (`counter++;`), whereas a `for` loop does the incrementation automatically.
- Note also the `if` code block indented inside the `do… while()` loop; the `if` statement is **nested** inside the loop.
- This is not the best way to write this program but does illustrate how several programming concepts can be combined.

An alternative to an array is a **list**. An array is thought of as a **static data structure** because it has a fixed size at run time. A list is a **dynamic data structure** that grows and shrinks to accommodate the data stored in it. Lists also contain additional methods and properties that allow you to interrogate or manipulate the list more easily than an array.

6. Note that below, the whole of Program.cs is shown. This is because to use Lists, you need to import the generic Collections toolkit, near the top (note the line numbers), before you can use Lists. Try entering the following program and see what happens:

```
1   using System;
2   using System.Collections.Generic;
3
4   namespace Objective_9
5   {
        0 references
6       class Program
7       {
            0 references
8           static void Main(string[] args)
9           {
10              // Create a new list
11              List<string> inventory = new List<string>();
12
13              //Add items to the list
14              inventory.Add("torch");
15              inventory.Add("gold coin");
16              inventory.Add("key");
17
18              //Remove items from the list
19              inventory.Remove("gold coin");
20
21              //Sort the items into alphabetical order
22              inventory.Sort();
23
24              //Output all the items in the list
25              foreach (string item in inventory)
26              {
27                  Console.WriteLine(item);
28              }
29          }
30      }
31  }
```

Type this in here first

This is the code you type in

7. Change the program so that a 'pick axe' is added to the inventory.

8. Change the program so the inventory is output and the user can choose which item to drop.

Objective 9: Key learning points

Understand arrays and lists

- Arrays are variables that have multiple elements, **all of the same data type**.
- Arrays are a **static data structure** because the size of the array does not change when the program is running.
- Lists are **dynamic data structures** because the size of the list changes when the program is running.
- An array or list has an **identifier**: the name of the array.
- An array or list has an **index** which points to one element of the array/list.
- Indexes start at 0.
- An array has multiple **elements**: the data stored in it.
- Arrays can be single dimension (one set of data) or be multi-dimension (having more than one set of data)
- Lists have **methods** that are operations that can be performed on the list with no additional code written by the programmer.

Objective 9: Key words

Array

Example code: `int[] x = new int[6];`

Creates a single dimension array called x with 6 elements (indexes 0-5).

`x[4] = 8;` would put the number 8 into the array x at index position 4.

Data can be populated into an array using braces and commas to separate items:

```
x = {4, 7, 3, 1, 9};
```

Multi-dimensional arrays are declared with maximum indexes separated with commas e.g.:

```
int [,] x = new int[10, 2];
```

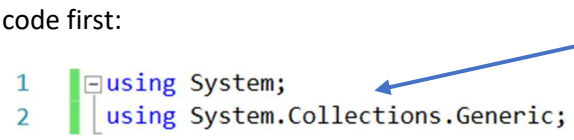
This can be visualised as a table with ten rows and two columns. It is possible to have up to 32 dimensions!

`x[4,1] = 8;` would put the number 8 into the array at row index 4 (5th row), column index 1 (2nd column); `y = x[6,1];` would put the value in row 6, column 1 into variable y.

List

Example code: `List<string> inventory = new List<string>();`

Creates a new list called 'inventory'. To use lists, you must import the collections toolkit at the top of the code first:



list.Add

Example code: `listname.Add(item);`

Adds item to the list.

list.Insert

Example code: `listname.Insert(index,item);`

Inserts item at position index.

list.Remove

Example code: `listname.Remove(item);`

Removes item from the list.

list.RemoveAt

Example code: `listname.RemoveAt(index);`

Removes an item from the list, stored at the index. Indexes start at 0 with the first item.

list.Count

Example code: `listname.Count;`

Outputs the number of items in the list.

list.Sort

Example code: `listname.Sort();`

Sorts all the items in a list.

list.Clear

Example code: `listname.Clear();`

Removes all the items from a list.

list.Contains

Example code: `if (listname.Contains(x))`

Returns whether the list contains item x as a Boolean.

Objective 10:
Understand serial files

In this objective you learn how to read and write data to files so it can be saved and loaded again later.

Tasks

1. Try entering the following program and see what happens:

You need to import the System.IO tools first. IO = Input / Output

Change "C:\\Learning C Sharp\\" to your own folder location. Datafile.txt is the name of the text file.
NOTE the double \\. This is to prevent confusion with escape characters such as \n.

```
using System;
using System.IO;

namespace Objective_10
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // Open a text file for writing
            string path = "C:\\Learning C Sharp\\datafile.txt";
            StreamWriter writer = new StreamWriter(path);

            // Write data to the file
            writer.WriteLine("This is a simple way to save data");
            writer.WriteLine("one line at a time");

            // Empty buffer & close the file
            writer.Close();
            Console.WriteLine("File Created");
        }
    }
}
```

Open the file `datafile.txt` and look at its contents.

Note the `path` variable contains the path and the filename. If the file does not exist, it creates it. You will need to change this so the file is written to your My Documents (or other) folder.

Note how `StreamWriter writer = new StreamWriter(path);` specifies that we are writing to the file. Another mode is append for adding to the end of an existing file. To do this, you add 'true' to the path parameter:

```
StreamWriter writer = new StreamWriter(path, true);
```

2. Try entering the following program and see what happens:

```
// Open a text file for reading
string path = "C:\\Learning C Sharp\\datafile.txt";
string data;
StreamReader reader = new StreamReader(path);

// read data from file
data = reader.ReadLine();
Console.WriteLine(data);
data = reader.ReadLine();
Console.WriteLine(data);

// Empty buffer & close file
reader.Close();
Console.WriteLine("File Closed");
```

Remember to change your file path if necessary

3. It is good practice to check if the file exists first. Try this:

```
// Open a text file for reading
string path = "C:\\Learning C Sharp\\noFile.txt";
string data;
if (File.Exists(path))  // Check if the file exists
{
    StreamReader reader = new StreamReader(path);
    // read data from file
    data = reader.ReadLine();
    Console.WriteLine(data);
    data = reader.ReadLine();
    Console.WriteLine(data);
    // Empty buffer & close file
    reader.Close();
    Console.WriteLine("File Closed");
}
else
{
    Console.WriteLine("File {0} does not exist.", path);
}
```

Remember to change your file path if necessary

4. Change the program so that 6 lines of text are written to a file and read back in again. Hint: more efficient to use loops.

It is often useful to read in all the data from a file line by line until you reach the end of the file. To do this we would want to combine an iteration with a file read.

5. Try entering the following program and see what happens:

```
// Open a text file for reading
string path = "C:\\Learning C Sharp\\datafile.txt";
StreamReader r = new StreamReader(path);
string line;
do
{
    // Read data from the file
    line = r.ReadLine();
    Console.WriteLine(line);
}
// Check if end of file
while (line != null); // 'null' means nothing there

// Close the file
r.Close();
```

Remember to change your file path if necessary

Objective 10: Key learning points

Understand serial files

- **Serial files** store data with no order to the data maintained.
- To search data from a serial file you begin at the start of the file and read all the data until the item is found.
- Data cannot be deleted from a serial file without creating a new file and copying all the data except the item you want to delete.
- Data cannot be changed in a serial file without creating a new file, copying all the data across to a new file, inserting the change at the appropriate point.
- Data can be **appended** (added to the end) to a serial file.
- A file can be open for reading or writing, but not reading and writing at the same time.
- Serial files are quite limiting, but are useful for simple data sets and configuration files. Other types of files include: sequential files where order of the data is maintained, index sequential files for large data sets and random files which allow you to access any item without searching through the file from the start.

Objective 10: Key words

System.IO

Example code: `using System.IO;`

Imports the data import and export tools which needs doing at the start of the code.

StreamWriter

Example code: `StreamWriter w = new StreamWriter(path);`

Creates a new file writing stream, called w. Path refers to the file path and name, including the file extension such as .txt. If the file does not exist, it creates it.

WriteLine()

Example code: `w.WriteLine(x);`

Writes a single item of data (x) to an open file, w.

StreamReader

Example code: `StreamReader r = new StreamReader(path);`

Creates a new file reading stream, called r. Path refers to the file path and name, including the file extension such as .txt.

ReadLine()

Example code: `x = r.ReadLine();`

Reads a single line of data from an open file, r, into x.

Close()

Example code: `x.Close();`

Commits any data in the file buffer to storage and closes the file, x.

File.Exists()

Example code: `if (File.Exists(path))`

Used to check if a file, path, exists or not. Can help reduce 'File not found' errors

null

Example code: `while (line != null);`

Null means nothing. Used to see if there is nothing in the next line of the file to determine if the end of file has been reached.

Objective 11:

How to handle exceptions for validation

In this objective you learn how to prevent your program crashing if it receives invalid data.

Tasks

1. Try entering the following program and use the test table of data to see what happens:

```
// Run time errors
decimal num1, num2, answer;
Console.WriteLine("Enter a number:");
num1 = Convert.ToDecimal(Console.ReadLine());
Console.WriteLine("Enter a number:");
num2 = Convert.ToDecimal(Console.ReadLine());
answer = num1 / num2;
Console.WriteLine("{0} divided by {1} = {2}", num1, num2, answer);
```

Test	First number	Second number	Expected
1	6	2	3.0
2	8	7	1.142
3	4	0	Crash – divide by zero
4	5	b	Crash – invalid character entered

A program should not crash because it has bad data input. The potential for errors should be considered by the programmer.

2. Try entering the following program and use the test table of data to see what happens:

```
// Run time errors
decimal num1, num2, answer;
num1 = 0;
num2 = 0;
// Get numbers
try
{
    Console.WriteLine("Enter a number:");
    num1 = Convert.ToDecimal(Console.ReadLine());
    Console.WriteLine("Enter a number:");
    num2 = Convert.ToDecimal(Console.ReadLine());
}
catch
{
    Console.WriteLine("Invalid entry");
}
// Output
try
{
    answer = num1 / num2;
    Console.WriteLine("{0} divided by {1} = {2}", num1, num2, answer);
}
catch
{
    Console.WriteLine("Unable to divide: divide by 0 error");
}
```

Objective 11: Key learning points

How to handle exceptions for validation

- There are 3 types of errors in programming:
  - **Syntax errors** when you mistype a keyword, the program doesn't recognise it as a valid command and will not run.
  - **Logic errors** where the program runs but you get the wrong result, perhaps not always consistently! Logic errors are called **bugs**.
  - **Run-time errors** that occur in exceptional circumstances such as division by zero and incorrect data type input. These should be trapped with **exception handling** commands.
- Fixing errors in code is known as **debugging**.
- Preventing invalid data input is known as **validation**. There are a number of different types of validation including:
  - **Presence checks**: did the user actually enter any data?
  - **Range checks**: does the input fall within a valid range of numbers?
  - **Format check**: does the input match a recognised format, e.g. LLNN NLL for postcode
  - **Length check**: is the input the required length of characters, e.g. minimum 8 character password.
- Most validation checks can be performed with selection and iteration statements, but sometimes an error could occur if one data type is converted to another, e.g. a string to an integer if the character is not a number.
- The development environment may contain a variety of tools to assist you in debugging a program:
  - Highlighting incorrect key words.
  - Compiler output errors that tell you why your program won't run.
  - **Break points** to stop the program at a particular point to check the value of variables.
  - **Stepping** through the program line by line to see what command is executing.
  - **Tracing** the value of variables as they step through the program.
  - An immediate window where you can try commands and see the output.

Objective 11: Key words

try… catch…

Example code:

```
try { x = y/z; }

catch { Console.WriteLine("Division by zero"); }
```

'Try' is the keyword to put before a run time error is likely to occur. The code following 'Catch is the code to execute if an error occurs.

There are may variations on try… catch. Research for further information here and this webpage to see an alternative to Convert.ToInt32 called TryParse.

Taking things further with C#

This has just been in introduction to C#. There is a lot more you can do; C# is a widely used and powerful programming language. There are many resources online for you to use but here are some links to get you started:

C# tutorial links from the AQA website:

- Home and Learn C# programming tutorial.
- Official Microsoft documentation for C#, including other links to tutorials.
- C# Wikibooks entry with tutorials and further information.

Other links you may find helpful / interesting:

- W3schools C# tutorial
- https://www.tutorialspoint.com/csharp/index.htm
- https://www.tutorialsteacher.com/csharp/csharp-tutorials
- This is a 4-hour video course: https://www.youtube.com/watch?v=GhQdlIFylQ8
- This is a full 24-hour video! https://www.youtube.com/watch?v=wfWxdh-_k_4
- Harvard introduction to game programming with Unity and C#.
- More on using Unity.