//

// *text*

Examples:

```
// This is a comment
```

Double forward slash: A comment.

Comments are ignored by the computer and discarded when a program is *translated* into *machine code*. They are used by programmers to explain the purpose of sections of code. This is helpful when you return to a program after a period of time, or when you work in teams.

Comments are typically used:

- At the beginning of a program to explain its purpose.
- Before each method.
- Before each selection statement: if, else, switch, case.
- Before each iteration: for, while.
- To explain difficult to comprehend code.
- To remind the author why unusual approaches have been taken.

For multiline comments, use /* */ like so:

```
/* first line of text
second line of text */
```

Associated keywords: `static, if, switch, case, while, do / while, for`

ADD

list.Add(*parameter*);

Examples:

```
my_list.Add("Dave")
```

Method: Adds the parameter to the end of a list.

The example above adds the string "Dave" to the list called my_list. The parameter can be a value or variable. The new item is always added to the end of the list and becomes the last item.

You can overwrite existing elements of a list by referring to their index. E.g. my_`list[2] = "Dave"` will replace what is currently stored in index 2.

Associated keywords: `Insert, RemoveAt, Remove`

ARRAY.INDEXOF

int variable = Array.IndexOf(arrayIdentifier, searchItem);

Examples:

```
string[] names = {"Craig", "Dave"};

int index = Array.IndexOf(names, "Dave");
```

Method: Returns the index of where the search item can be found in an array.

For example, if an array `myArr` contained: `["A", "B", "C", "D"]` then `Array.IndexOf(myArr, "C")` would return 2 because the item "C" is stored at index 2. Remember that arrays are zero-indexed which means the first item is stored at index 0. The method only returns the first occurrence of an item in an array.

If the item is not in the array this method will return -1. This could be used to check if an item exists within an array and negates the need for a searching algorithm.

If you want to find more than one occurrence in the array, you should consider using a search algorithm instead. For example, a linear search that uses a for loop to iterate over all the items in an array.

ARRAY.REVERSE

Array.Reverse(*arrayIdentifier*, *optionalStartIndex*, *optionalLength*);

Examples:

```
Array.Reverse(myArray);
```

```
Array.Reverse(myArray, 1, 4);
```

Method: Reverses the sequence of elements in a one-dimensional array.

If only a subset of the array needs to be reversed, use the `optionalStartIndex` and `optionalLength` parameters. The `optionalStartIndex` parameter specifies the start of the subset, and the `optionalLength` parameter specifies the number of elements of the subset to reverse.

If the method is run without specifying a start index and length, then the entire array will be reversed.

This is useful if you want the items in a list in descending order. Use `Array.Sort()` to initially sort the items and then reverse the order with `Array.Reverse()`.

Associated keywords: `Array.Sort`

ARRAY.SORT

Array.Sort(*arrayIdentifier*);

Examples:

```
Array.Sort(myArray);
```

Method: Sorts the items in an array into ascending or alphabetical order.

C# uses an *introspective sort algorithm* to order the items in an array.

Associated keywords: `Array.Reverse`

COMPARETO

StringExp.CompareTo(StringExp)

Examples:

```
"dave".CompareTo("craig")
```

```
if ("harry".CompareTo("anna") == -1) {}
```

Method: returns -1 if the Unicode value of the first string is less than the second's Unicode value, 0 if the first string is equal to the second, or 1 if the Unicode value of the first string is greater than the second's Unicode value.

As C# does not allow strings to be compared using < or >, the `CompareTo` method must be used. The `CompareTo` method can also be used for numeric types, but it may be easier to use comparison operators for these instead.

For example, `"a".CompareTo("b")` would return -1, because the Unicode value of "a" is 97, which is less than the Unicode value of "b", which is 98.

It can be used as a condition in selection statements.

Associated keywords: `if, switch`

CONSOLE.READLINE

variable = Console.ReadLine(*parameter*);

Examples:

```
Console.Write("Enter your forename: ");
string forename = Console.ReadLine();
```

Method: returns an input from the keyboard.

Inputs from the keyboard allow user interaction with your program. Inputs are always sequences of alphanumeric characters called *strings* terminated when the user presses the enter key.

The `Console.Write()` line is optional but informs the user what data they are expected to enter, so often it is used directly before `Console.ReadLine()` is used. `Console.Write()` is used instead of `Console.WriteLine()` because this allows the user to type on the same line as the prompt.

It is common to add an extra space at the end of the prompt to separate the prompt and the input.

Remember to do calculations on the input, the input data will need to be *cast* to an *integer* or *double*.

CONSOLE.WRITE
CONSOLE.WRITELINE

Console.Write(*parameter*);

Console.WriteLine(*parameter*);

Examples:

```
Console.Write("Hello World")
Console.Write(x)
Console.WriteLine("Goodbye Venus")
Console.WriteLine(x)
```

Command: Outputs a value to the screen. The value can be any data type: Boolean, integer, list, float, string.

Don't forget that what you want to output must be enclosed in brackets. Strings will need to be qualified by quotes.

Console.Write() does not begin a new line after outputting its parameter to the screen, whereas Console.WriteLine() does. Often Console.Write() is used before using Console.Readline(), to inform the user of what they are expected to enter. E.g.

```
Console.Write("Enter a planet: ");
string planet = Console.ReadLine();
```

A single blank line, or the end of a line can be outputted with:

```
Console.WriteLine();
```

Multiple parameters of the same data type can be outputted on one line. Each parameter is separated with an addition symbol. The output will not include an automatic space between each parameter, so you will need to take this into account by adding blank spaces as shown:

```
print("You are " + age + " years old")
```

Associated keywords: Appendix: Interpolated strings

CONTAINS

list.Contains(*variable*);

Examples:

```
List<string> names = new List<string>() {"Craig ",
"Dave"};

if (names.Contains("Dave"))

if (y.Contains(x))

while (y.Contains(x))
```

Method: Returns whether a constant or a variable is contained within a list.

The `Contains` method is very useful for checking if an item exists within a list and negates the need for a searching algorithm. It can also be used to check whether a substring is in a string.

It can be used as a condition for selection and iteration commands.

Note that the `Contains` method is case sensitive.

Returns `True` if the variable is in the list or `False` if not.

Associated keywords: `if, while, do / while`

CONVERT.TODOUBLE

double variable = Convert.ToDouble(*parameter*)

Examples:

double x = Convert.ToDouble("5");

double y = Convert.ToDouble(6);

double z = Convert.ToDouble(a);

Method: Casts a parameter to a double (real) number.

Different types of data are stored in different ways by the computer. Inputs taken from the keyboard are always sequences of alphanumeric characters called *strings*. These sequences of characters are stored as numbers by the computer using a standard such as *ASCII* or *Unicode*. To do calculations on the input, it first needs to be converted from a string to a number, or from "5.6" to 5.6. This is known as *casting*.

Whole numbers with no fractional part are known as *integers*. Numbers with a fractional part (decimal places) are known as *doubles*, *real numbers*, *reals*, *floats* or *floating point numbers*.

A *run-time error* will occur if the parameter cannot be cast to a double.

Associated keywords: `Convert.ToInt32, Convert.ToString`

CONVERT.TOINT32

int *variable* = Convert.ToInt32(*parameter*);

Examples:

```
int x = Convert.ToInt32("5");
int x = Convert.ToInt32(6.5);
int x = Convert.ToInt32(y);
```

Method: Casts a parameter to the int data type, which can store whole numbers from $-(2^{31}-1)$ to $2^{31}-1$.

Different types of data are stored in different ways by the computer. Inputs taken from the keyboard are always sequences of alphanumeric characters called *strings*. These sequences of characters are stored as numbers by the computer using a standard such as *ASCII* or *Unicode*. The input string "5" (strings are *qualified* by quotes) is stored as the number 53. To do calculations on the input, it first needs to be converted from a string to a number, or from 53 to 5. This is known as *casting*.

Whole numbers with no fractional part are known as *integers*. Numbers with a fractional part (decimal places) are known as *floating points*, *floats*, *real numbers* or *reals*. Using `Convert.ToInt32` to cast a float will round the float using *banker's rounding* to the nearest integer, and remove the fractional component.

Banker's rounding is where midpoint values are rounded to the nearest even number. For example, both 3.75 and 3.85 round to 3.8.

It's worth noting that if a character is passed into this method, it will be converted to the ASCII value of the character. For example, 'a' will be converted to 97 when passed into this method.

Associated keywords: `Convert.ToDouble, Convert.ToString`

COUNT

parameter.Count;

Examples:

```
int x = myList.Count;
```

Property: Returns the number of indexes in the parameter.

Remember the number of indexes includes the first index which is index 0.

The parameter can be a list in which case the number of elements is returned. This command can also be used for various collection types in C#, such as dictionaries.

Associated keywords: `Length`

DO / WHILE

```
do
{
        // code block to be executed
}
while (condition);
```

Examples:

```
bool valid_input = false;
do
{
      Console.Write("Enter your choice ");
      string choice = Console.ReadLine();
}
while (!valid_input);
```

```
int i = 1;
do
{
  Console.WriteLine(i);
```

```
  i++;
}
while (i < 11);
```

Command: repeats the indented section of code until the condition is met.

This loop will execute the code block first, then check if the condition is true, then it will repeat the code inside the loop for as long as the condition is true.

Code to be executed must be inside the curly brackets and indented. This is often the source of many logic errors, so check your code is indented correctly.

Repeated sections of code are known as iterations or loops. Use a do/while command when it is not known in advance how many iterations will be required.

It is good practice to comment before this command to explain the purpose of the iteration.

More than one condition can be combined with logic operators and brackets can be used to group conditions.

It is possible to include another do/while command within an indented section. This is known as nesting.

Associated keywords: `//, for, foreach, while`

FOR

```
for (initialisation; condition; change)
{
        // code block to be executed
}
```

Examples:

```
for (int counter = 0; counter < 5; counter++)
{
      Console.WriteLine(counter);
}

for (int counter = 5; counter >= 0; counter--)
{
      Console.WriteLine(counter);
}
```

Command: Repeats the indented section of code a given number of times.

The *initialisation* statement is executed first and only once. Usually, it sets a variable before the loop starts. The *condition* statement defines the condition for the loop to run – if it is true, the loop will repeat the

code inside it, otherwise the loop will end. The *change* statement changes the variable initialised in the *initialisation* statement at the end of every iteration of the loop.

Code to be executed must be inside the curly brackets. This is often the source of many logic errors, so check your code is correct.

Repeated sections of code are known as *iterations* or *loops*. Use a for command when you want to repeat a known number of times.

It is good practice to comment before this command to explain the purpose of the iteration.

It is possible to include another for loop within an for loop. This is known as *nesting*.

Although the functionality of a for command can be replicated with a while command it is usually considered good practice to use a for loop for finite iterations.

As a for command will always iterate a finite number of times, to maximise the efficiency of an algorithm you should consider if a while loop or an alternative command could be used instead to terminate the iteration early. It is possible to terminate a for loop before it is complete with a break command, but this is not considered good practice because it creates more than one exit point for the command which increases the complexity of testing. The use of break should usually be avoided.

Associated keywords: `//`, `foreach`, `while`, `do / while`

FOREACH

foreach (*type variable* in *array*)

{

// code block to be executed

}

Examples:

```
string[] names = {"Craig", "Dave"};
foreach (string name in names)
{
  Console.WriteLine(name);
}
```

Command: Loops through each element of an iterable data type (such as an array, list, or string), with each iteration giving `variable` the value of the next element in the iterable data type.

This can also be achieved with a for loop, although it is generally considered that a foreach loop is more readable and easier to understand.

It is good practice to comment before this command to explain the purpose of the iteration.

Associated keywords: `//`, `for`, `while`, `do / while`

GETLENGTH

arrayIdentifier.GetLength(*dimension*);

Examples:

```
my_array.GetLength(0)
```

Method: returns the number of elements in the specified dimension of the array.

For example, my_array.GetLength(0) will return the number of elements in the first dimension of the array.

The dimension specified must be zero-based which means the first dimension is stored at dimension 0.

If the dimension specified is less than 0, or greater than or equal to the number of dimensions in the array, an exception will occur.

Associated keywords: `Length`

IF

```
if (condition)
{
        // code block to be executed
}
else if (condition)
{
        // code block to be executed
}
else
{
        // code block to be executed
}
```

Examples:

```
if (x == y)
{
      Console.WriteLine("The value of x is the same
as y");
}
else if (x < y)
{
```

```
      Console.WriteLine("The value of x is less
than y");
}
else
{
      Console.WriteLine("The value of x is greater
than y");
}
```

Command: Selects which code branch to execute next depending on the outcome of a condition.

The condition requires two variables or constants to be compared with mathematical or logic operators (see appendix). More than one condition can be combined with logic operators and brackets can be used to group conditions.

E.g.

```
if ((x > y) && (x > 6)) {}
```

Note that you should not use `if (x > y > 6)` as it will not work as you expect. Instead, be explicit about which **two** items of data are being compared in **each** condition.

The result of an `if` command is always either true or false.

That means you can also use a shorthand for Boolean conditions. E.g.
`if (valid) {}` is the same as `if (valid == true) {}`
`if (!valid) {}` is the same as `if (valid == false) {}`

The `else if` section is an optional part of the command to include alternative conditions. You can include as many additional else if sections as you need but consider using the command `switch` instead if you require more than one else if section.

The `else` section is an optional part to execute if none of the conditions are met, including those in `else if` sections.

Code to be executed for each section must be inside the curly brackets. This is often the source of many logic errors, so check your code is in the correct place.

It is good practice to comment each section of this command to explain the purpose of each condition.

It is possible to include another if command within an indented section. This is known as *nesting*.

Associated keywords: `//, switch`

INSERT

list.Insert(*index*, *parameter*);

Examples:

```
myList.Insert(2, "Dave");
```

Method: Inserts the parameter into a list at the specified index.

The example above inserts the string "Dave" into the list called myList at index 2. The indexes of the existing items at index 2 and above are incremented. The parameter can be a value or a variable.

You can overwrite existing elements of a list by referring to their index. E.g. `myList[2] = "Dave"` will replace what is currently stored in index2.

Associated keywords: `Add, Remove, RemoveAt`

LENGTH

int *variable* = *identifier*.Length;

Examples:

```
int x = "ABC".Length;
```

```
int x = my_array.Length;
```

Property: returns the number of indexes in the object specified.

Remember the number of indexes includes the first index which is index 0.

The parameter can be a string, in which case the number of characters is returned, or an array in which case the number of elements is returned.

If the array is multidimensional, `Length` gives the total number of elements across all dimensions. To get the number of elements in a specific dimension, `GetLength` can be used.

Associated keywords: `GetLength, Count`

MATH.ABS

Math.Abs(*parameter*);

Examples:

```
int x = Math.Abs(-5);
```

```
int x = Math.Abs(y);
```

Method: Returns the absolute value of the parameter.

The absolute value is a positive value. For example, the absolute value of -6 is 6. This method is useful for turning a negative number into a positive number.

It can also be useful to swap a number from positive to negative or negative to positive. This can be achieved with `x = -x`.

MATH.CEILING

Math.Ceiling(*parameter*);

Examples:

```
double x = 65.23;
x = Math.Ceiling(x);
```

Method: Rounds the parameter up to the nearest integer.

In the example above x would be assigned as 66.

Associated keywords: `Math.Floor, Math.Round`

MATH.FLOOR

Math.Floor(*parameter*);

Examples:

```
double x = 65.83;

x = Math.Floor(x);
```

Method: Rounds a number down to the nearest integer.

In the example above x would be assigned 65.

Associated keywords: `Math.Ceiling, Math.Round`

MATH.PI

double *variable* = Math.PI;

Examples:

```
double x = Math.PI;
```

Field: Returns the constant pi as 3.1415926535897931

MATH.POW

Math.Pow(base, exponent);

Examples:

```
double x = Math.Pow(8, 2);
```

Method: Returns the result of the first parameter (the base) raised to the power of the second parameter (the exponent).

Both parameters are required, if you don’t specify each of them then you'll encounter a compilation error in your code.

In the example above, 8 will be raised to the power of 2, which will give the result of 64.

MATH.ROUND

Math.Round(*requiredParameter1*, *optionalParameter2*, *optionalParameter3*)

Examples:

```
double x = Math.Round(6.532234, 2);
```

```
double x = Math.Round(y, 3,
MidpointRounding.AwayFromZero);
```

Method: Rounds a number to the nearest integer or to a particular number of decimal places.

The first parameter must be the number to round. If this number is a double, this method will return a double. Likewise, if this number is a decimal, this method will return a decimal.

The second parameter specifies the number of decimal places in the output. By default, this is 0.

The third parameter specifies the rounding strategy to be used. These include `MidpointRounding.AwayFromZero`, where Midpoint values are rounded to the next number away from zero. For example, 3.75 rounds to 3.8 and 3.85 rounds to 3.9.

Another rounding technique is `MidpointRounding.ToEven`, where midpoint values are rounded to the nearest even number. For example, both 3.75 and 3.85 round to 3.8. By default, this is `MidpointRounding.ToEven`.

The first parameter is required, while the second and third parameters are optional.

Associated keywords: `Math.Ceiling, Math.Floor`

MATH.SQRT

Math.Sqrt(*parameter*)

Examples:

```
int x = Math.Sqrt(25);
```

```
x = Math.Sqrt(y);
```

Method: Returns the square root of a specified number.

This method takes one parameter of type double. A positive (double) value will be returned, unless the parameter is negative or NaN (not a number), in which case NaN will be returned. If the parameter is PositiveInfinity, the function will return PositiveInfinity.

NEXT

variable.Next(*optionalParameter, optionalParameter*)

Examples:

```
Random random_generator = new Random();

int x = random_generator.Next();

int y = random_generator.Next(100);

int dice = rand.Next(1, 6);
```

Method: Returns a random integer. A random number generator must be instantiated before using it, as shown in the example section above. If neither of the optional parameters are specified, a non-negative random integer will be returned.

If one of the parameters is specified, this parameter is the upper boundary of the random number to be generated. A non-negative random integer that is less than the value of this parameter will be returned. If this parameter is specified as a negative number, an exception will be raised.

Specifying both parameters will return a random integer that is within a range. The first parameter will be the (inclusive) lower bound of the random number returned, and the second parameter will be the (exclusive) upper bound of the random number returned – it must be greater than or equal to the lower bound. Any parameters specified should be integers.

Associated keywords: `Random`

RANDOM

Random *variable* = new Random(*optionalSeed*);

Examples:

```
Random random_generator1 = new Random();
```

```
Random random_generator2 = new Random(1);
```

Constructor: Initialises a new instance of the Random class using a seed value.

Computers cannot generate random numbers because they can only perform calculations. Instead, they use a calculation on a number known as a seed to generate what looks like a random number to the user. For example, the fractional part of the square root of 55 is 4161984871. If you didn't know the algorithm and the seed value of 55, these digits would appear to be random.

If a value for the seed is not specified, the time of day will be used by default. Specifying a value will ensure the random number function always generates the same deterministic sequence of numbers. The seed value should be an integer, and if a negative seed value is used, the absolute value of the number is used.

This constructor must be used before trying to generate random numbers.

Associated keywords: `Next`

REMOVE

list.Remove(*parameter*);

Examples:

```
myList.Remove("Dave");
```

Method: Removes the first occurrence of the parameter from a list.

If the element is found and removed, the method returns `true`, otherwise it returns `false`.

The example above removes the string "Dave" from the list called myList. Remember that this will reduce the index of all other elements stored after this index by -1, as well as reducing the size of the list by -1.

To remove all instances of the parameter without an error you would need to use a while loop. E.g.

```
while (myList.Contains("Dave")) {
    myList.Remove("Dave");
}
```

Associated keywords: `Add, Insert, RemoveAt`

REMOVEAT

list.RemoveAt(*index*);

Examples:

```
myList.RemoveAt(2);
```

Method: Removes an element from a list at a specified index.

The example above removes the item stored at index 2 from the list called myList. Remember that this will reduce the index of all other elements stored after the index by -1, as well as reducing the size of the list by -1.

Associated keywords: `Add, Insert, Remove`

RETURN

return *expression*

Examples:

```
static int square (int x)
{
      return x * x;
}
y = square(5);
```

Command: Returns a value from a method.

Subprograms that return values are called *functions*.

In the example above, the number 5 is passed into the method called square and assigned to the parameter x. The variable is then multiplied by itself and returned as the output from the function square into the variable y which assigns it the value 25.

The return expression can be a Boolean, e.g. return True, a variable, e.g. return total, a list or the result of a calculation. However, you must return an expression of the same type that is named in the subprogram definition.

If you need to return more than one value, you will need to return a list or array.

Methods that do not return a value are known as *void methods*.

REVERSE

list.Reverse(*optionalStartIndex*, *optionalLength*);

Examples:

```
myList.Reverse();
```

```
myList.Reverse(1, 4)
```

Method: Reverses the items in a list.

If only a subset of the list needs to be reversed, use the *optionalStartIndex* and *optionalLength* parameters. This allows you to specify the starting index and number of elements from that starting index of the list to reverse.

If the method is run without specifying a start index and length, then the entire list will be reversed.

This is useful if you want the items in a list in descending order they've been sorted in ascending order. Use `.Sort()` to initially sort the items and then reverse the order with `.Reverse()`.

Associated keywords: `Sort`

SORT

list.Sort();

Examples:

```
myList.Sort();
```

Method: Sorts the items in a list into ascending order.

C# uses a *Quick Sort* to order the items in a list.

Associated keywords: `Reverse`

STATIC

static *type identifier* (*parameters*) {}

Examples:

```
static int square (int x)
{
      x = x * x;
      return x;
}
```

Command: Defines a new Method. Methods are also called subprograms and *subroutines*.

Code must be contained within the curly brackets. You cannot use spaces in the identifier name of the method. It is common to use underscores to separate words in the name of the method instead.

Don't forget the opening curly bracket at the end of this command, and the closing curly bracket after you have finished defining the method's code.

Methods can be *void*, meaning that they do not return a value.

Methods are used to structure a program into smaller more manageable parts. This is known as *problem decomposition*. If your method is a procedure that does not return a value, then instead of writing a data type in the definition, write the word *void* instead.

Methods are used to create reusable program components. If your method is a function which returns a value, then you'll need to specify the data type which it returns in the method definition, where we've written *type*.

Methods avoid unnecessary code duplication and make the code easier to read which also makes finding errors in code, called *debugging* easier.

Associated keywords: `return`

SWITCH

switch(*variable*)
{
case *value*:
// code block
break;
default:
// code block
break;
}

Examples:

```
switch (day_name)
{
      case "Thu":
      case "Fri":
      case "Sat":
            Console.WriteLine("Open 10-5pm");
            break;
      case "Sun":
```

```
            Console.WriteLine("Open 11-3pm");
            break;
      default:
            Console.WriteLine("Closed.");
            break;
}
```

Statement: An alternative to the if/else if/else structure that is used to select one of many code blocks to be executed. Switch is considered a better command to use than the if/else if/ else structure when there are many outcomes because it is more readable for multiple values. Don't forget the colon after each case.

You can have as many case statements as you need, and each one should include a comment.

It works by first evaluating the switch expression once, then comparing the value of the expression with the values of each case. If there is a match, the associated block of code is executed.

The `break` keyword is used at the end of each case block so that when the match is found, and code execution inside of that block is finished, it breaks out of the switch block.

`default:` is the equivalent to `else` and captures any value that was not matched.

Associated keywords: `//, if`

USING

using *namespace;*

Examples:

```
using System;
```

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

Command: Includes additional namespaces in your program.

In C#, namespaces are used to organise and group related objects. For example, the System namespace includes fundamental objects, such as `Console` and `Math`.

The `using` command allows you to access objects in different namespaces.

Remember that to use lists in your program, you have to make sure the `System.Collections.Generic` namespace is included in your program by writing `using System.Collections.Generic` at the top of your program.

WHILE

while (*condition*)

{

// code block to be executed

}

Examples:

```
bool valid_input = false;
while (!valid_input)
{
      Console.Write("Enter your choice: ");
}
while (choice < 0 || choice > 3)
{
      Console.Write("Enter your choice: ");
}
```

Command: Repeats the indented section of code until the condition is not met.

Code to be executed must be inside the curly brackets. This is often the source of many logic errors, so check your code is in the right place.

Repeated sections of code are known as *iterations* or *loops*. Use a while command when it is not known in advance how many iterations will be required.

It is common to ensure the condition cannot be met before the first iteration to ensure the indented code executes at least once.

It is good practice to comment before this command to explain the purpose of the iteration or condition.

More than one condition can be combined with logic operators and brackets can be used to group conditions.

It is possible to include another if commands within an indented section. This is known as *nesting*.

While loops are often used with indented input commands for *validation*, ensuring that the user has entered a valid input before continuing the program.

A special value that uses its presence as a condition to terminate a loop is called a *sentinel value*.

Infinite loops can be created with `while (true)` since true will always be true.

Associated keywords: `//, for, foreach, do / while`

Appendix 1

Variable assignment

```
type identifier;
```

```
type identifier = value;
```

You can use the above syntax to define a variable, where type is the data type of the value it holds, and identifier is the variable name – this cannot contain any spaces, so often underscores are used to separate words instead. You may declare a variable without giving it any contents (as shown in the first example), or by using an equals sign and specifying an initial value. The different data types are listed below.

`int`	A whole number, without any decimal places
`double`	A floating-point number or decimal
`char`	A single character, wrapped in single quotes.
`string`	A combination of characters stored as text, wrapped in double quotes
`bool`	One of two states: true or false

To change the value stored in a variable after its declaration, you do not need to state the type. Just state the variable's identifier and the new value you would like it to store.

```
identifier = value;
```

Constants

```
const type identifier = value;
```

If you don't want the value of a variable to be changed or overwritten, you can use the const keyword in front of the variable type. Once you have done this, the variable will be read-only, and its contents cannot be changed later in the program.

You cannot declare a constant variable without assigning the value.

String manipulation

Concatenation

```
string x = "Hello" + " " + "World"
```

To concatenate means to join together.

Numbers should be cast to strings before they are concatenated.

Creating strings of characters

It is possible to create strings of repeated characters using the following syntax.

```
string variable = new string('@', 5);
```

The variable would be assigned "@@@@@". The first parameter is the character which is to be repeated, the second parameter is the number of times that it should be repeated.

Comparison operators

==	`if x == y`	Is x the same as y? (equal)
!=	`if x != y`	Are x and y different? (not equal)
<	`if x < y`	Is x less than y?

<=	`if x <= y`	Is x less than or equal to y?
>	`if x > y`	Is x greater than y?
>=	`if x >= y`	Is x greater than or equal to y?

Note that a double equal is asking a question, a single equal assigns a variable. E.g.

`x == 6` means is x equal to 6?

`x = 6` means x becomes the number 6.

Logical operators

and	`if x > y && x > 6:`	Both conditions must be true for the result to be True.
or	`if x > y || x > 6:`	One of the conditions must be true for the result to be True.
not	`if ! x:`	The condition must not be met for the result to be True.

Mathematical operators

+	`x = 6 + 5`	Addition
-	`x = 6 - 5`	Subtraction
*	`x = 6 * 5`	Multiplication
/	`x = 6 / 5`	Division *Floating-point division is performed when the data type is a double.*
**	`Math.Pow(6, 5)`	Exponentiation *See the dedicated page in the main body of this guide for details on how to use Math.Pow.*
%	`x = 5 % 5`	Modulus

Interpolated strings

```
string name = "Dave";

string message = $"Hello, {name}";

string result = $"The sum of {x} and {y} is {x + y}.";
```

Interpolated strings allow you to format a string for output. As an alternative to concatenation, interpolation provides more options to embed expressions and manipulate variables used in the output.

To create an interpolated string, use the $ symbol before quotation marks. You can then include expressions inside curly brackets {} within the string.

Interpolation can also be used to format numerical values. This is often by writing `:[format specifier][precision specifier]` after the value to format. The format specifier is an alphabetical character which specifies the type of number format, for example currency. The precision specifier is optional, and specifies the number of decimal places the number should be formatted to.

The below table contains six common format specifiers, what they are used for and example usage.

C	Currencies	`$"{4.5675:C2}"`	`$4.47`
F	Fixed point numbers	`$"{43.896:F2}"`	`43.90`
N	Numbers – adds in separators	`$"{1758:N2}"`	`1,748.00`
P	Percentages	`$"{0.3986:P2}"`	`39.86 %`
D	Decimals – adds in leading zeros	`$"{34:D4}"`	`0034`
E	Exponential (scientific) format	`$"{1234.56:E2}"`	`1.23E+003`

1D arrays

An array is a data structure that holds a collection of elements of the same type. Each element in the array has an index, which is its position in the array, and each element can be accessed by its index.

Arrays are zero-indexed, meaning that the index of the first element in an array is 0, the second element is 1, and so on.

Once an array is created, its size cannot be changed.

To declare a new 1D array, you can use the following syntax:

elementDataType[] *identifier* = new *elementDataType*[*arrayLength*];

For example, the below would declare a 1D array to store 5 integers:

```
int[] array1 = new int[5];
```

Once an array is declared, you can then insert values into it.

You can also declare a new 1D array and set array element values in one line, which could be useful if the array is not very long.

elementDataType[] *identifier* = [*element1*, *element2*, …];

For example, the below would declare an array of the numbers from one to five.

```
int[] array2 = [1, 2, 3, 4, 5];
```

1D lists

A list is a collection of elements of the same data type. Like arrays, each element has an index representing its position in the list.

Lists are dynamic, meaning that their size can change as a program is running. They are part of the Systems.Collections.Generic namespace, so to use lists in a program you have to include it:

```
using Systems.Collections.Generic;
```

To declare a new list, you can use the following syntax:

List<*elementDataType*> *identifier* = new List<*elementDataType*>();

For example, the below would declare and initialise a new list of names:

```
List<string> names = new List<string>;
```

You can then add elements to this list using commands such as **Add**.

If you already know some elements which you want to add, you can initialise the list with these elements:

List<*elementDataType*> *identifier* = new List<*elementDataType*> {*element1*, element2, … };

For example, you could create a list of names like so:

```
List<string> names = new List<string> {"Craig",
"Dave"};
```

Lists are dynamic, so it is possible to add further elements to the list.

Arrays vs Lists

It can be hard to know whether to use an array or a list in your program.

Remember that lists are dynamic (their length can change), whereas arrays have a fixed size. For example, an array might be suitable for storing days of the week, as its length (7) is fixed and known in advance. A list might be more suitable for recording user inputs, where the number of inputs is not known in advance.

Lists have more built-in methods, such as `Add` and `Remove`, which can make it easier to work with collections of data. However, arrays are generally more memory-efficient than lists, so if your collection size is fixed and performance is a priority, an array may be more suitable.

It can be confusing to know which commands are for lists and which are for arrays. The below table shows a brief description of common commands, and the list and array commands.

Description	List keyword	Array keyword
Check if an element is in a collection	`.Contains();`	Use `Array.IndexOf();` – returns -1 if element is not in array.
Insert an element	`.Insert();`	n/a
Find the length of a collection	`.Count;`	`.Length;`
Remove an element at a certain index	`.RemoveAt();`	n/a
Remove an element	`.Remove();`	n/a
Find the position of an element	`.IndexOf();`	`Array.IndexOf();`
Reverse the elements	`.Reverse();`	`Array.Reverse();`
Sort the elements	`.Sort();`	`Array.Sort();`
Add an element	`.Add();`	n/a

Shuffling elements in an array

If you want to shuffle the elements in an array, it's recommended to create a helper function as shown below:

```
static string[] shuffle(string[] list_to_shuffle)
{
    int n = list_to_shuffle.Length;
    while (n > 1)
    {
        n--;
        int k = random_generator.Next(n + 1);
        string value = list_to_shuffle[k];
        list_to_shuffle[k] = list_to_shuffle[n];
        list_to_shuffle[n] = value;
    }
    return list_to_shuffle;
}
```

This method uses the Fisher-Yates (also known as Knuth) shuffle algorithm. This algorithm starts from the last element of an array, and then randomly selects an element from the unshuffled portion of the array. These elements are swapped, and this process repeats until all the elements have been shuffled.

Command Index