# //

## // *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 */
```

# 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: Interpolated strings

# 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`

# 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*.

# 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`

# 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.

## 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` |

## Mathematical operators

| | | |
|---|---|---|
| + | x = 6 + 5 | Addition |
| - | x = 6 - 5 | Subtraction |
| * | x = 6 * 5 | Multiplication |
| / | x = 6 / 5 | Division<br>*Floating-point division is performed when the data type is a double.* |

# Command Index