Expressions and Control Flow

Let's start with an example of what we've seen so far:

```
using System;

namespace AddingNumbers
{
    class Program
    {
        static void Main(string[] args)
        {
            int a, b;
            Console.WriteLine("Sum of two interers");
            Console.Write("a = ");
            a = int.Parse(Console.ReadLine());
            Console.Write("b = ");
            b = int.Parse(Console.ReadLine());
            Console.WriteLine("The sum of " + a + " and " + b + " is " + (a + b));
            Console.ReadKey(true);
        }
    }
}
```

On the first line of our Main function we declare the two variables that will hold the values that we're going to add together. The next line outputs to the standard output a short informative description of what we're about to do.

Next we urge our user to enter a value for our first addend "a". Then we read a line from the standard output and we pass it to the Parse function of the int type. This will convert the string we have read from the standard input into an integer value and store it into the variable "a". We do the same for "b".

The last two lines output the sum of "a" and "b". Note that the "+" operation behaves differently. When we use it on a string and an integer the integer value is implicitly converted to string and then the two strings are concatenated. This is the behavior we have in:

```
"The sum of " + a
```

On the other side the next expression will follow operator priorities – it will first execute the expression in the brackets (add "a" and "b"). The result of the operation will be an integer value that will be concatenated to the " is " string.

```
" is " + (a + b)
```

Another way to output the result would be to create an additional variable "c" and store the result from adding "a" and "b" there as per the next example:

```
int c = a + b;
Console.WriteLine("The sum of " + a + " and " + b + " is " + c);
```

The ReadKey function at the last line of code will keep the program running until a key from the keyboard is pressed. This way the console will stay open and we will be able to see the program's console output.

Let's take a look at another example. This time we're calculating the distance by a given time period and constant speed.

```
using System;

namespace Distance
{
    class Program
    {
        static void Main(string[] args)
        {
            float speed, time, distance;

            Console.WriteLine("Calculating path");
            Console.Write("Enter the speed (m/s): ");
            speed = float.Parse(Console.ReadLine());
            Console.Write("Enter the time (s): ");
            time = float.Parse(Console.ReadLine());

            distance = speed * time;

            Console.WriteLine("The distance is (m): " + distance);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we declare the three floating point variables – "speed", "time" and "distance". The next line outputs to the standard output a short informative description of what we're about to do.

Next we urge our user to enter the constant speed. We read a line from the console and then use the Parse function of the float type to convert the string value we have read from the console into a floating point value. We do the same for the time.

On the next line we calculate the distance by the formula distance equals speed times time and we output the distance to the standard output.

We can format the output in the following manner:

```
Console.WriteLine("The distance is (m): " + distance.ToString("F3"));
```

Or

```
Console.WriteLine("The distance is (m){0:F3}: ", distance);
```

The second way to format the output is by using the built-in formatting functionality in the WriteLine function. Here {0:F3} contains the sequential number of the expression or variable after the formatting string (in our case 0 – the first argument after the formatting string) and a

standard or custom formatter for the value (in our case F3 which stands for floating point number with 3 digits after the floating point.

Operators

The C# programming language provides several types of operators. Let's start with arithmetic operators. They are used on numerical values:

Operator	Description
+	When used with numerics adds values together. When used with strings concatenates two strings.
-	Substracts two values.
*	Multiplies two values
/	Divides two values.
%	Computes the remainder after dividing its first operand by its second.

Next let's take a look at logical operators. They are used on boolean values:

Operator	Description
&&	This operator is the logical AND. The result is true only when the two operands are true.
||	This operator is the logical OR. The result is false only when the two operands are false.

Unary operators have only one operand. In C# the built-in unary operators are increment and decrement operators:

Operator	Description
++	The incrementation operator is an unary operator. It is used on numerics. It increments the operand with 1.
--	The decrementation operator is an unary operator. It is used on numerics. It decrements the operand with 1.

Relational operators are used for comparing values.

Operator	Description
==	Equals returns true when the two operands have equal values.
!=	Not equal returns true when the two operands have different values.
>	Greater than returns true when the left operand has greater value than the right.
<	Less than returns true when the right operand has greater value than the left.
>=	Greater or equal returns true when the left operand has greater or equal value than the right.
<=	Less or equal than returns true when the right operand has greater or equal value than the left.

In order to understand better operators let's write a program that calculates the area of a circle with a given radius:

```
using System;

namespace CircleArea
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159;
            double r;
            Console.Write("Enter a radius (cm): ");
            r = double.Parse(Console.ReadLine());
            Console.WriteLine("The area of a circle with radius {0:F3} cm is {1:F3} cm²",
                              r, pi * r * r);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we declare the constant "pi". On the second line we declare a double variable "r" which will hold the radius of the circle. Next we urge our user to enter the radius, read it, convert it from string to double and store it in "r".

On the last two lines we output a string explaining what we've calculated. We use the built-in formatting functionality in the WriteLine function. The first argument we format as a floating point number with three digits after the floating point. This is the radius. The second argument is the area that we calculate. Note that the multiplication of values will be executed and then the result will be passed as a parameter to the WriteLine function.

Let's try another problem – converting temperature from Celsius to Fahrenheit and Kelvin:

```
using System;

namespace Temperature
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Celsius to Fahrenheit and Kelvin");
            //    freeze    step   boiling
            // C    0        1       100
            // F   32       1.8      212
            // K  273.15     1      373.15
            Console.Write("Enter the temperature in degrees (Celsius): ");
            float Cd = float.Parse(Console.ReadLine());
            float Fd = Cd * 1.8f + 32;
            float Kd = Cd + 273.15f;
            Console.WriteLine("Fahrenheit temperature is {0:F2}", Fd);
            Console.WriteLine("Kelvin temperature is {0:F2}", Kd);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we output a short informative description of what we're about to do. Note the next four lines. They start with `//` which is the comment symbol. All lines within a C# program that are prefixed with double slashes are treated as code comments and are ignored by the compiler.

On the next line we urge the user to enter the temperature that is to be converted in Celsius. Then we read a line from the standard output and convert it to a float value. On the next two lines we declare two variables "Fd" and "Kd" – the temperature in Fahrenheit and Kelvin. The formulas can be taken out of the comments. On the next two lines we output the results properly formatted.

Expressions

An expression in a programming language is an executable combination of function calls and operations over constants, variables and values. These combinations usually obey a fixed list of implicit operation priorities. In the general case priorities can be specified explicitly by placing sub-expressions in brackets.

Let's introduce priorities with a more complex example. We're going to take a look at a program for calculating the distance between two points in a plane. Say we have point A with coordinates xA and yA , and point B with coordinates xB and yB . The distance is $\sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$:

```
using System;

namespace Distance
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter the coordinates of the first point (cm):");
            Console.Write("x = ");
            double xa = double.Parse(Console.ReadLine());
            Console.Write("y = ");
            double ya = double.Parse(Console.ReadLine());
            Console.WriteLine("Enter the coordinates of the second point (cm):");
            Console.Write("x = ");
            double xb = double.Parse(Console.ReadLine());
            Console.Write("y = ");
            double yb = double.Parse(Console.ReadLine());

            double distance = Math.Sqrt(Math.Pow(xa - xb, 2) + Math.Pow(ya - yb, 2));

            Console.WriteLine("The distance between two points is {0:F4}", distance);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we output a short informative description of what we're about to do. Next we urge the user to enter a value for the x coordinate of the starting point. We declare the variable "xa", read a line from the standard input, convert it to double and then assign it to the "xa" variable. We do the same for the y coordinate of the starting point. On the next five lines we output a string that informs the user that we are about to ask him the coordinates of the second point. Then we read its x and y coordinates and store them in the "xb" and "yb" variables.

On the next line we declare the "distance" variable and assign it with the computed distance between the two points.

Here you can see we're using the square root function of the built-in mathematical functions in the .NET Framework. Note that the sum is supplied as a parameter to the square root function. Let's trace the evaluation of the expression according to priorities.

The first two sub-expressions that are going to be evaluated are the subtraction of "xa" and "xb" and "ya" and "yb". Next the results are going to be raised to the power of two and then added together. Then the Sqrt function will calculate the square root of the result.

The next line outputs the distance in a formatted manner.

The built-in mathematical functions in the .NET Framework are accessible through the Math class. Here is a list of the most commonly used mathematical functions:

Function	Description
Abs	Returns the absolute value of a number.
Ceiling	Returns the smallest integral value that is greater than or equal to the specified decimal number.
Cos	Returns the cosine of the specified angle.
Floor	Returns the largest integer less than or equal to the specified decimal number.
Max	Returns the larger of two numeric values.
Min	Returns the smaller of two numeric values.
Pow	Returns a specified number raised to the specified power.
Round	Rounds a double-precision floating-point value to a specified number of fractional digits.
Sin	Returns the sine of the specified angle.
Sqrt	Returns the square root of a specified number.
Truncate	Calculates the integral part of a specified floating point numeric value

Operation priorities

Operation priorities differ based on the operators. Let's take a look at the usage of logical operators. In the example below we declare four Boolean variables. To "cond1" we assign the value true. To "cond2" – false. To "p1" we assign the result of the 7 > 5 expression which evaluates to true. To "p2" – the result of the 7 < 5 expression which evaluates to false.

Let's take a look at the code:

```
using System;

namespace LogicalOperators
{
    class Program
    {
        static void Main(string[] args)
        {
            bool cond1 = true;
            bool cond2 = false;
            bool p1 = 7 > 5;
            bool p2 = 7 < 5;
            Console.WriteLine("7 > 5 is " + p1);
            Console.WriteLine("7 < 5 is " + p2);
            Console.WriteLine(cond1 + " and " + cond2 + " is " + (cond1 && cond2));
            Console.WriteLine(cond1 + " or " + cond2 + " is " + (cond1 || cond2));
            Console.ReadKey(true);
        }
    }
}
```

On line five and six we output the values of "p1" and "p2". They will be implicitly casted to strings before the concatenation. On line seven "cond1" and "cond2" will also be casted to string before concatenation. Note that if we remove the brackets of the (cond1 && cond2) the compiler will see it as a syntax error because && is not applicable to string values. That is because the concatenation operator ("+") has higher priority than the logical and operator ("&&"). On the next line we do a logical "or".

Control Flow statements

In programming the Control Flow refers to the order in which individual statements are executed. In imperative programming languages the Control Flow can be determined runtime by control flow statements. These statements use control flow operators that determine the control flow of the program based on logical expressions. Let's take a look at a program that determines the maximum of two numbers:

```
using System;

namespace Maximum
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 8, b = 9;
            int max = a > b ? a : b;
            Console.WriteLine("Max value is " + max);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we declare two integer variables “a” and “b” and assign them values.

The next line introduces the triple operator. Its syntax is:

```
condition ? expression : expression
```

The condition is a logical expression used to direct the control flow. The first expression is evaluated and returned if the condition evaluates to true. If the condition evaluates to false then the second expression is evaluated and returned. The triple operator returns the value of the evaluated expression.

In our case if “a” is greater than “b” then “a” is returned. If not then “b” is returned. The result of the triple operator is the bigger of the two numbers. The value is then outputted to the standard output.

Another way to do that is by using the “if” statement:

```
using System;

namespace Maximum
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 8;
            int b = 9;
            int max;

            if (a > b)
            {
                max = a;
            }
            else
            {
                max = b;
            }

            Console.WriteLine("Max value is " + max);
            Console.ReadKey(true);
        }
    }
}
```

Here the “max” variable is declared but is not assigned. Then we use the “if” statement to direct the control flow. If “a” is greater than “b” then “max” is assigned with the value of “a”. If “a” is not greater than “b” then “max” is assigned with the value of “b”. The next line outputs the maximum value to the standard output.

We use the “if” statement to direct the control flow. In the syntax below you can see that there are three key points in the “if” statement – the condition, the consequent code block and the alternative code block.

The condition is a logical expression. Its result determines the control flow of the program. If the condition evaluates to true the "if" statement will execute the consequent code block. In the cases where the condition evaluates to false the "if" statement will execute the alternative code block. The "else" section of the "if" statement can be omitted.

```
if (condition)
{
  // consequent code block
}
else
{
  // alternative code block
}
```

The else statement can be used multiple times in combination with if statement in the following manner:

```
if (condition_1)
{
  // consequent code block
}
else if (condition_2)
{
  // alternative code block 1
}
else if (condition_3)
…
```

In these cases if condition_1 is not met then the "if" statement checks condition_2 and so on. When one of the conditions is met the "if" statement skips the remaining "else if" sections.

In C# the code blocks are surrounded with curly brackets. In the cases where the code block consists of only one statement the brackets can be omitted. Let's have an example with the previous problem. Since the consequence and alternative code blocks both consist of one statement the brackets can be omitted as per the following code:

```
if (a > b)
    max = a;
else
    max = b;
```

Let's continue with a more complex example. Say we have three values and want to determine the maximum among them. In the last example we had to compare two values thus the code paths were two. Now we have to compare three values and the code paths are four. Let's take a look at the code:

```
using System;

namespace Maximum
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 7, b = 5, c = 8, max;

            if (a > b)
            {
                if (a > c)
                    max = a;
                else
                    max = c;
            }
            else
            {
                if (b > c)
                    max = b;
                else
                    max = c;
            }

            Console.WriteLine("Max value is " + max);
            Console.ReadKey(true);
        }
    }
}
```

On the first line of the Main function we declare four integer variables. We will compare “a”, “b” and “c” and store the maximum in “max”. Note that we now need to nest “if” statements in order to go through all the cases.

On the second line we compare “a” and “b”. On the next line we have a nested “if” statement. If we’re executing this if statement then we already know that “a” is bigger than “b” (the condition of the parent “if” statement is obviously met). If “a” is bigger than “b” and “a” is bigger than “c” then “a” is the maximum thus we assign “max” with the value of “a”.

If the a > c condition is not met we execute the “else” section. In this case “a” is bigger than “b” but “a” is not bigger than “c” then obviously “c” is the maximum thus we assign “max” with the value of “c”.

In the “else” section of the main “if” statement we proceed analogically. In the end we output the value of max to the standard output.

When talking about programming there are always easy, hard, complex and simple ways to do things. Let’s cut down the lines of code we need to write in order to compare the three values.

We need to increase the complexity of the conditions so that we can obtain more information from them. The “a” holds the maximum value when it is larger than “b” and is larger than “c”. We can make a logical “AND” of the two conditions that “a” needs to satisfy as per line

one of the example below. If “a” is not the maximum it’s either “b” or “c” so we need a simple comparison between them to identify which is larger as in the “else if” section. If “b” is larger than “c” then the maximum is “b”. The only other possibility is left for the “else” section.

```
if (a > b && a > c)
    max = a;
else if (b > c)
    max = b;
else
    max = c;
```

Let’s continue with a simple program that salutes us depending on the current time.

```
using System;

namespace Salute
{
    class Program
    {
        static void Main(string[] args)
        {
            int hour;
            Console.Write("Enter the hour: ");
            hour = int.Parse(Console.ReadLine());

            string greeting;

            if (hour < 5)
                greeting = "Good night!";
            else if (hour < 12)
                greeting = "Good mornig!";
            else if (hour < 18)
                greeting = "Good afternoon!";
            else if (hour < 22)
                greeting = "Good evening!";
            else
                greeting = "Good night";

            Console.WriteLine(greeting);
            Console.ReadKey(true);
        }
    }
}
```

If the current time is between 10PM and 5AM then the program should output “Good night”. If we’re between 5AM and 12PM then it’s “Good morning”. If it’s past noon but before 6PM it’s “Good afternoon” and if it’s past 6PM but before 10PM it’s “Good evening”.

On the first line of the Main function we declare an integer that will hold the current hour. Next we urge the user to enter the current hour, convert it and assign it to the “hour” variable.

Next we declare the “greeting” string variable. In the “if” statement we check the value of “hour” and assign appropriate greeting string to the “greeting” variable. At the end we output the greeting string.

We can modify the program so that it will take the current time from the system clock. This is done through the DateTime type.

```
int hour = DateTime.Now.Hour;
```

The DateTime.Now obtains the date and time of the system clock. It exposes the current second, minute, hour, day, month and year. As you've seen so far we access the nested elements within a complex structure by the "." symbol.

Let's take a look at the control flow operator "switch". The next program will ask our user for the temperature in Celsius, Fahrenheit or Kelvin and will output the temperature in all three scales.

In the first line of the Main function in the code below we urge the user to enter the temperature that is to be converted across the three scales. We read a line from the standard input and then convert it to a decimal value. Next we need the user to specify whether the temperature is in Celsius, Fahrenheit or Kelvin. We urge the user to enter a letter C, F or K. On the next line we declare a character variable "kd", read a single character using the Read function of Console. We cast the value that we have read into a char value and assign it to the character variable.

Next we define three constants that will help us convert temperature. They are the water freezing temperature of the Fahrenheit scale and the relation between 1 degree Celsius and 1 degree Fahrenheit and the water freezing temperature of the Kelvin scale. Next we declare three variables that will hold the result (the temperature in the three scales) and initialize them with 0. On the next line we create a Boolean variable that will tell us whether the input is correct. Now that that's out of the way we can proceed to the actual conversion.

We have three code paths based on the scale type of the input temperature (Celsius, Fahrenheit or Kelvin). Here we use the switch / case control flow construct. On the first line of the "switch" construct we define the expression that will determine which "case" block will be executed (the "kd" variable has the scale of the input temperature). Each "case" block is defined with a constant value. If the constant value is equal to the expression from the "switch" statement then this case block is executed. If a case statement does not have a code block then the execution falls through and the next "case" block is executed (its constant value is not taken into account). Every "case" code block ends with a "break" statement. The break statement ends the execution of the "switch" construct.

The first "case" block of our "switch" construct handles the code path where the input is in Celsius. The temperature is converted to the other two scales. Note that we cover both upper and lower case letters by fall through "case" sections.

The "default" section of the "switch" construct is executed when there is no matching "case". In our program if we reach the "default" section our input is incorrect. In that case we output informative message and assign false to the "correct_input" variable.

If the “correct_input” variable has value “true” the input is correct and we output the temperature converted in all three scales.

```
static void Main(string[] args)
{
    Console.Write("Enter the temperature: ");
    double temp = double.Parse(Console.ReadLine());
    Console.Write("Enter the letter C (Celsius), F (Fahrenheit), K (Kelvin): ");
    char kd = (char)Console.Read();
    const double f_step = 1.8;
    const double f_freezing = 32;
    const double k_freezing = 273.15;
    double C_degree = 0, F_degree = 0, K_degree = 0;
    bool correct_input = true;

    switch (kd)
    {
        case 'C':
        case 'c':
        {
            C_degree = temp;
            F_degree = C_degree * f_step + f_freezing;
            K_degree = C_degree + k_freezing;
            break;
        }
        case 'F':
        case 'f':
        {
            F_degree = temp;
            C_degree = (F_degree - f_freezing) / f_step;
            K_degree = C_degree + k_freezing;
            break;
        }
        case 'K':
        case 'k':
        {
            K_degree = temp;
            C_degree = K_degree - k_freezing;
            F_degree = C_degree * f_step + f_freezing;
            break;
        }
        default:
        {
            Console.WriteLine("You have entered another letter!");
            correct_input = false;
            break;
        }
    }
    if (correct_input)
    {
        Console.WriteLine("The temperature is");
        Console.WriteLine("in Celsius: {0:F2}", C_degree);
        Console.WriteLine("in Fahrenheit: {0:F2}", F_degree);
        Console.WriteLine("in Kelvin: {0:F2}", K_degree);
    }
    Console.ReadKey(true);
}
```

To end this chapter let's have another example – a program that adds, subtracts, multiplies or divides numeric values.

```
using System;

namespace Temperature
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter a real number: ");
            double a = double.Parse(Console.ReadLine());

            Console.Write("Enter second real number: ");
            double b = double.Parse(Console.ReadLine());

            Console.Write("Select one operation +, -, * or /  ");
            char op = (char)Console.Read();

            double result = 0;

            switch (op)
            {
                case '+':
                    result = a + b;
                    break;
                case '-':
                    result = a - b;
                    break;
                case '*':
                    result = a * b;
                    break;
                case '/':
                    result = a / b;
                    break;
            }

            if (op == '+' || op == '-' || op == '*' || op == '/')
            {
                Console.WriteLine(a.ToString("F2") + op + b.ToString("F2") + " = " +
                                  result.ToString("F2"));

                Console.WriteLine("{0:F2}{1}{2:F2} = {3:F2}", a, op, b, result);
            }

            Console.ReadKey(true);
        }
    }
}
```

In order to be able to do that our program has to read from the user the two elements of the operation (floating point numeric values) and the operation type (addition, subtraction, multiplication or division).

On the first line of the Main function we urge our user to enter a real number (the left operand). We then read the value from the standard input, convert it to double and store it in a newly declared variable “a”. On the next two lines we do the same for the right operand.

Next is the operation itself. We urge the user to enter the operation by outputting a line to the standard output with the available operations that are supported by our program. On the next line we declare a character variable “op” that will hold the operation type, we read a character from the standard input, cast it to char and assign it to our “op” variable. On the next line we declare a variable named “result” of type double that will hold the result of the operation.

On the next line we start our “switch” construct. When the value of the “op” variable is the plus sign we add the two decimals (“a” and “b”) and assign the result to the “result” variable.

The next “case” section of our “switch” will be executed when the value of the “op” variable is the minus sign. Then we subtract the two operands and again assign the result to the “result” variable.

The “case” blocks for multiplication and division work analogically.

The line after the switch construct checks the validity of the operation. This time we don’t do it in a “default” section of the “switch” construct but in an “if” statement. If the operation is equal to at least one of the operations that are supported by our program then we output the result in a formatted manner. If not we don’t output anything.

Now that we’ve seen two examples of the switch / case construct we can see its syntax:

```
switch(expression)
{
    case constant_value_1:
    {
        code_block_1
        break;
    }
    case constant_value_2:
    {
        code_block_2
        break;
    }
    …
    default:
    {
        default_code_block
    }
}
```

During the execution of the program the expression is evaluated and its result is compared with the constant values in the “case” blocks. When a “case” block with a “constant_value” equal to the expression result is found its “code_block” is executed. When the “break” statement is executed the “switch” construct stops execution. The brackets of the “case” code block can be omitted.

The “code_block” of a “case” section can be omitted. In these cases the execution falls through to the next “case” section and its code block is executed instead. This technique is used when we a case to be executed if the expression matches at least one value from a given set. An example is a switch construct with the week days where we want to identify which part of the week is it:

```
using System;

namespace Week
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Today is: ");
            string today = Console.ReadLine();

            switch (today)
            {
                case "monday":
                case "tuesday":
                case "wednesday":
                case "thursday":
                case "friday":
                {
                    Console.WriteLine("Working day");
                    break;
                }
                case "saturday":
                case "sunday":
                {
                    Console.WriteLine("Weekend");
                    break;
                }
            }

            Console.ReadKey(true);
        }
    }
}
```

In the example above if the “today” variable has the name of one of the working days the code block of the “friday” “case” section will be executed. If the value is a weekend day then the code block of the “sunday” “case” section will be executed.