C#

STEP BY STEP PROGRAMMING GUIDE

A DETAILED APPROACH TO PRACTICAL CODING

NATHAN CLARK

C#

A Detailed Approach to Practical Coding

Nathan Clark

This document is presented with the desire to provide reliable, quality information about the topic in question and the facts discussed within. This book is sold under the assumption that neither the author nor the publisher should be asked to provide the services discussed within. If any discussion, professional or legal, is otherwise required a proper professional should be consulted.

This Declaration was held acceptable and equally approved by the Committee of Publishers and Associations as well as the American Bar Association.

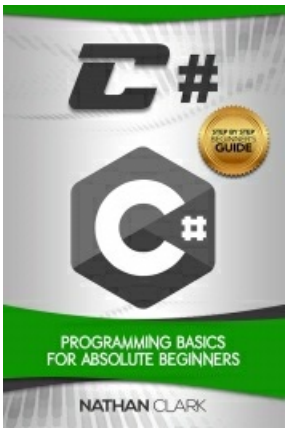
Any information contained in the following pages is considered accurate and truthful and that any liability through inattention or by any use or misuse of the topics discussed within falls solely on the reader. There are no cases in which the Publisher of this work can be held responsible or be asked to provide reparations for any loss of monetary gain or other damages which may be caused by following the presented information in any way shape or form.

The following information is presented purely for informative purposes and is therefore considered universal. The information presented within is done so without a contract or any other type of assurance as to its quality or validity.

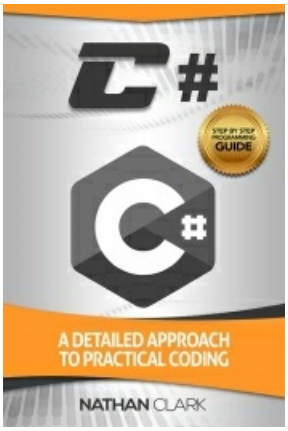
Any trademarks which are used are done so without consent and any use of the same does not imply consent or permission was gained from the owner. Any trademarks or brands found within are purely used for clarification purposes and no owners are in anyway affiliated with this

work.

Books in this Series



C#: Programming Basics for Absolute Beginners



C#: A Detailed Approach to Practical Coding

Table of Contents

Introduction

This book is the second book in the Step-By-Step C# series. If you haven't read the first book C#: Programming Basics for Absolute Beginners, I highly suggest you do so before getting into this book.

In this intermediate level guide we will delve more into further concepts of C#. With each topic we will look at a detailed description, proper syntax and numerous examples to make your learning experience as easy as possible. C# is a wonderful programming language and I trust you will enjoy this book as much as I enjoyed writing it. So without further ado, let's get started!

1. Loops

The usage of loops helps us to iterate through a set of values. Let's say you wanted to go through a list of students and display their names; loops can help you iterate through the list of students easily and can help access each record separately.

Loops work on a condition and only execute the code based on that condition. A simple view of how this works is shown below.

```
Loop(condition)
{
//Execute code
}
```

So in the above abstract code snippet, we can see that the code will be executed based on the evaluation of the condition in the loop statement. There are different types of loop statements and in this chapter we will go through each of the available loops in more detail.

1.1 While Loops

The general syntax of the while loop is given below.

```
While(condition)
{
//execute code
}
```

As long as the condition is true in the while loop, the code will keep on executing in the while code block. Let's look at an example of the while statement.

Example 1: The following program is used to showcase how to use the while loop.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int i = 1;

         // While loop
         while (i <= 5)
         {
            Console.WriteLine("The value is " + i);
            i++;
         }

         Console.Read();

      }
   }
}
```

Things to note about the above program:

- We are defining an integer ‘i’ which has an initial value of 1.
- In the while loop we state the condition that while the value of ‘i’ is less than 5, keep on executing the code in the while code block.
- In the while code block, we display the value of ‘i’ and also increment the value of i.

With this program, the output is as follows:

The value is 1

The value is 2

The value is 3

The value is 4

The value is 5

1.2 Do-While Loops

The general syntax of the do-while loop is given below.

```
do
{
//execute code
}
While(condition);
```

Here the difference between the do-while loop and the normal while loop is that the condition is tested at the end of the block of code. This means that you will always be guaranteed that the block of code will be executed at least once. Let's look at an example of the do-while statement.

Example 2: The following program is used to showcase how to use the do-while loop.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int i=1;

         // do-While loop
         do
         {
            Console.WriteLine("The value is " + i);
            i++;
         }
         while (i <= 5);

         Console.Read();
```

```
        }
    }
}
```

With this program, the output is as follows:

The value is 1

The value is 2

The value is 3

The value is 4

The value is 5

1.3 For Loops

The general syntax of the for loop is given below:

```
for(initialization;condition;incrementer)
{
//execute code
}
```

In the for clause, you can specify the initialization, condition and incrementer in one statement. Let's look at an example of the for loop statement.

Example 3: The following program is used to showcase how to use the for loop.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         // for loop
         for(int i=0;i<5;i++)
         {
            Console.WriteLine("The value is " + i);
         }

         Console.Read();

      }
   }
}
```

Things to note about the above program:

- We are now initializing the value of ‘i’ in the for loop itself.
- We are also testing for the condition of the value of ‘i’ in the for loop itself.
- Next we are incrementing the value of ‘i’ in the for loop as well.

With this program, the output is as follows:

The value is 0

The value is 1

The value is 2

The value is 3

The value is 4

1.4 Nested Loops

We can also nest loops one inside of another. Let's look at some examples of nested loops to best explain this concept.

Example 4: The following program is used to showcase how to use nested loops.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         // for loop
         for(int i=0;i<3;i++)
         {
            Console.WriteLine("The value of i is " + i);
            for (int j = 0; j < 2; j++)
            {
               Console.WriteLine("The value of j is " + j);
            }
         }

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of i is 0

The value of j is 0

The value of j is 1

The value of i is 1

The value of j is 0

The value of j is 1

The value of i is 2

The value of j is 0

The value of j is 1

Let’s see another example on how we can use nested loops.

Example 5: The following program is used to showcase how to use nested loops.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         // for loop

         for(int i=0;i<3;i++)
         {
            Console.WriteLine("The value of i is " + i);
            int j = 0;
            while(j<2)
            {
               Console.WriteLine("The value of j is " + j);
               j++;
            }
         }

         Console.Read();

      }
```

```
    }
}
```

With this program, the output is as follows:

The value of i is 0

The value of j is 0

The value of j is 1

The value of i is 1

The value of j is 0

The value of j is 1

The value of i is 2

The value of j is 0

The value of j is 1

2. Decision Making in C#

The usage of decision making loops helps us to execute code only if a particular condition holds true.

Let says we wanted to search for records in a student database and only award scholarships to those students whose aggregate marks were above 90%, then we can use decision loops for this purpose. A sample decision loop structure is shown below.

```
if(condition)
{
//Execute code
}
```

So in the above abstract code snippet, we can see that the code will be executed based on the evaluation of the condition in the if statement. Only if the condition is true will the code block statements be executed.

There are different types of decision making statements and in this chapter we will go through each of the available options in more detail.

2.1 If Statements

In the if statement we get the chance to perform an action only if a certain condition evaluates to true. The general syntax of the if statement is given below.

```
if(condition)
{
//Execute code
}
```

Example 6: The following program is used to showcase how to use the if statement.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int i=3;
         // if statement

         if(i==3)
            Console.WriteLine("The value of i is " + i);

         Console.Read();

      }
   }
}
```

Things to note about the above program:

- We are defining a condition in the if statement, which says that only if the value of ‘i’ is 3 will we write to console that the value of i is 3.

With this program, the output is as follows:

The value of i is 3

2.2 If-Else Statement

In the if-else statement we have the option of executing an optional statement when the if condition does not evaluate to true. The general syntax of the if-else statement is given below.

```
if(condition)
{
//Execute code
}
else
{
//Execute code
}
```

Example 7: The following program is used to showcase how to use the if-else statement.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int i=4;
         // if statement

         if(i==3)
            Console.WriteLine("The value of i is " + i);
         else
            Console.WriteLine("The value of i is not equal to 3");
         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of i is not equal to 3

2.3 Switch Statement

The switch statement helps us to evaluate multiple options at one time and then execute code based on those various options. The general syntax of the switch statement is given below.

```
switch(expression) {
   case constant-expression  :
      statement(s);
      break;
   case constant-expression  :
      statement(s);
      break;

   default :
   statement(s);
}
```

In the switch statement, you can evaluate the condition against multiple expressions. For each matching expression you can have a corresponding statement to execute. Once the statement is found, the break statement helps to exit from the switch statement.

You can also have a default statement which gets executed if none of the expressions matches the conditions in the switch statement.

Example 8: The following program is used to showcase how to use the switch statement.

```
using System;

namespace Demo
{
    // A simple application using C#

    class Program
    {
        // The main function
        static void Main(string[] args)
        {
```

```
            int i=4;
            switch (i)
            {
               case 1: Console.WriteLine("The value of i is 1");
                  break;

               case 2: Console.WriteLine("The value of i is 2");
                  break;

               case 3: Console.WriteLine("The value of i is 3");
                  break;

               case 4: Console.WriteLine("The value of i is 4");
                  break;

               default: Console.WriteLine("The value of i is unknown");
                  break;
            }

            Console.Read();

         }
      }
}
```

Things to note about the above program:

- We are defining the switch statement to evaluate the value of ‘i’.
- We are defining the different possible values of ‘i’ using the case statements.
- We finally define the default statement which gets executed if none of the case statements match the desired expression.

With this program, the output is as follows:

The value of i is 4

2.4 Nested Statements

We can also make use of nested statements. Let's look at some examples of how nested statements can be implemented.

Example 9: The following program is used to showcase how to use nested decision making statements.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int i=4;

         if (i > 0)
         {
            if (i == 4)
            {
               Console.WriteLine("The value is 4");
            }
         }

         Console.Read();

      }
   }
}
```

In the above program we are making use of multiple if statements in the program.

With this program, the output is as follows:

The value of i is 4

Let’s look at another example of using nested decision making statements.

Example 10: The following program is used to showcase how to use nested decision making statements.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int i=4;

         if (i > 0)
         {
            switch (i)
            {
               case 1: Console.WriteLine("The value of i is 1");
                  break;

               case 2: Console.WriteLine("The value of i is 2");
                  break;

               case 3: Console.WriteLine("The value of i is 3");
                  break;

               case 4: Console.WriteLine("The value of i is 4");
                  break;

               default: Console.WriteLine("The value of i is unknown");
                  break;
            }
         }

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of i is 4

3. Methods

Methods are used to define a set of logical statements together. These sets of statements are then used for a definitive purpose. For example, if you wanted to add a set of numbers together, you can define a method for that. Then call the method whenever you want to add numbers. The general syntax of a method is shown below.

```
Returndatatype methodname(parameters)
{
//Block of code
return datavalue
}
```

1. A method can have an optional return type. If the method is going to return a value to the main calling program, you can define the type of the data value being returned.
2. You can also define optional parameters which can be sent to the method. This is useful for making the method more dynamic in nature.
3. Next, you have the return statement return the data value if a value needs to be returned by the method.

Let's now look at a simple example of how to define a method and call the method accordingly.

Example 11: The following program is used to showcase how to use methods.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
```

```
        // Method definition
        static void Add()
        {
            int i=3;
            int j=4;

            Console.WriteLine("The sum of the integers is "+ (i + j));

        }
        // The main function
        static void Main(string[] args)
        {

            // calling the method in the main program
            Add();
            Console.Read();

        }
    }
}
```

In the above program we are:

- Defining a method called ‘Add’ which has the purpose of adding 2 integer numbers.
- Next we call the ‘Add’ method in the main calling program.

With this program, the output is as follows:

The sum of the integers is 7

Now let’s look at an example of how to pass parameters to a method.

Example 12: The following program is used to showcase how to use methods with parameters.

```
using System;

namespace Demo
{
    // A simple application using C#

    class Program
```

```
    {
        // Method definition
        static void Add(int i,int j)
        {

            Console.WriteLine("The sum of the integers is "+ (i + j));

        }
        // The main function
        static void Main(string[] args)
        {

            // calling the method in the main program
            Add(3,4);
            Console.Read();

        }
    }
}
```

In the above program we are making the method more dynamic by having the ability to pass any set of the values to the method.

With this program, the output is as follows:

The sum of the integers is 7

Now let's look at an example of how to pass parameters to a method and also return a value to the main calling program.

Example 13: The following program is used to showcase how to use methods with parameters and a return type.

```
using System;

namespace Demo
{
    // A simple application using C#

    class Program
    {
        // Method definition
        static int Add(int i,int j)
```

```
        {

            return (i + j);

        }
        // The main function
        static void Main(string[] args)
        {

            // calling the method in the main program
            Console.WriteLine("The sum of the integers is " + Add(3, 4));
            Console.Read();

        }
    }
}
```

In the above program we are:

- Defining the ‘Add’ method to return a datatype of the type integer.
- We then use the return statement to return the sum to the main calling program.

With this program, the output is as follows:

The sum of the integers is 7

3.1 Recursive methods

You can also define recursive methods in which the method keeps calling itself iteratively. A good example of this is the implementation of the factorial series, which is shown via the example below.

Example 14: The following program is used to showcase how to use recursive methods.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // Method definition
      static int factorial(int num)
      {

         int result;
         if (num == 1)
         {
            return 1;
         }
         else
         {
            result = factorial(num - 1) * num;
            return result;
         }
      }
      // The main function
      static void Main(string[] args)
      {

         // calling the method in the main program
         Console.WriteLine("The factorial of 6 is " + factorial(6));
         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The factorial of 6 is 720

4. Arrays

Arrays are used to define a collection of values of a similar datatype. The definition of a data type is shown below.

```
datatype[] arrayname= new datatype[arraysize]
```

The arraysize is the number of elements that can be defined for the array. Let's now look at an example of how we can define an array in C#.

Example 15: The following program is used to showcase how to work with arrays.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {
         // Defining the array
         collect[0]=1;
         collect[1]=2;
         collect[2]=3;

         Console.Read();

      }
   }
}
```

Now with the above program:

- We are defining an integer array which has a size of 3.
- The first item in the array starts with the 0 index value.

- Each value of the array can be accessed with its corresponding index value.
- Each value of the array is also known as the element of the array.

Let's now look at an example of how we can display the values of the array.

Example 16: The following program is used to showcase how to work with arrays and display the elements of the array.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {
         // Defining the array
         int[] collect=new int[3];

         // Defining the elements of the array
         collect[0]=1;
         collect[1]=2;
         collect[2]=3;

         // Displaying the elements of the array
         Console.WriteLine("The first element of the array is " +
collect[0]);

         Console.WriteLine("The second element of the array is " +
collect[1]);

         Console.WriteLine("The third element of the array is " +
collect[2]);

         Console.Read();

      }
   }
```

```
}
```

With this program, the output is as follows:

The first element of the array is 1

The second element of the array is 2

The third element of the array is 3

We can also define an array of different data types other than the integer data type. Let's look at another example where we can define an array of strings.

Example 17: The following program is used to showcase how to work with string arrays.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {
         // Defining the array
         string[] collect = new string[3];

         // Defining the elements of the array
         collect[0]="First";
         collect[1]="Second";
         collect[2]="Third";

         // Displaying the elements of the array
         Console.WriteLine("The first element of the array is " +
collect[0]);

         Console.WriteLine("The second element of the array is " +
collect[1]);

         Console.WriteLine("The third element of the array is " +
```

```
collect[2]);

            Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The first element of the array is First

The second element of the array is Second

The third element of the array is Third

We can also carry out the normal operations with the elements of the array, just as we would normally do with ordinary variables. Let's look at an example on how we can work with the elements of the array.

Example 18: The following program is used to showcase how to work with array elements with standard operators.

```
using System;

namespace Demo
{
    // A simple application using C#

    class Program
    {
        // The main function
        static void Main(string[] args)
        {
            // Defining the array
            string[] collect = new string[3];

            // Defining the elements of the array
            collect[0]="First";
            collect[1]="Second";
            collect[2]="Third";

            int[] team = new int[3];
            team[0] = 1;
```

```
            team[1] = 2;
            team[2] = 3;

            //Working with arrays
            Console.WriteLine("The combination of the first array is " +
collect[0]+collect[1]+collect[2]);

            Console.WriteLine("The combination of the second array is " +
team[0] + team[1] + team[2]);

            Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The combination of the first array is FirstSecondThird

The combination of the second array is 123

4.1 Initializing Arrays

Arrays can also be initialized when they are defined. The syntax for the array definition is given below.

```
Datatype[] arrayname={value1,value2…valueN-1}
```

The values for the array are defined in the parenthesis { }. Let's now look at an example of this.

Example 19: The following program is used to showcase how to initialize arrays.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int[] team = { 1, 2, 3 };

         //Working with arrays

         Console.WriteLine("The combination of the array is " + team[0]
+ team[1] + team[2]);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The combination of the array is 123

4.2 Going through the Elements of the Array

When an array has a number of elements, you might need to use the loops available in C# to iterate through the elements of the array. Let's look at an example of this in action.

Example 20: The following program is used to showcase how to iterate through the elements of the array.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int[] team = new int[10];

         // Setting the values of the array
         for (int i = 0; i < 10; i++)
            team[i] = i;

         // Displaying the values of the array
         for (int i = 0; i < 10; i++)
         Console.WriteLine("The value of team[" + i + "] is " + team[i]);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of team[0] is 0

The value of team[1] is 1

The value of team[2] is 2

The value of team[3] is 3

The value of team[4] is 4

The value of team[5] is 5

The value of team[6] is 6

The value of team[7] is 7

The value of team[8] is 8

The value of team[9] is 9

The value of team[10] is 10

4.3 Multidimensional Arrays

One can also define multidimensional arrays, which have different dimensions. The most common form of a multidimensional array is the 2 dimensional array. The table below shows how an array is structured if the following hypothetical array is defined.

```
Collect[3,2]
```

Here, 3 is the number of rows and 2 is the number of columns. So the array can have a maximum of 6 values.

	Column 0	**Column 1**
Row 0	1	2
Row 1	3	4
Row 2	5	6

Example 21: The following program is used to showcase how to define a two dimensional array.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int[,] collect = new int[3,2];

         collect[0,0] = 1;

         collect[0,1] = 2;

         collect[1,0] = 3;
```

```
            collect[1,1] = 4;

            collect[2,0] = 5;

            collect[2,1] = 6;

            // Displaying the values of the array

            Console.WriteLine("Value at collect[0,0]  is " + collect[0,0]);
            Console.WriteLine("Value at collect[0,1]  is " + collect[0,1]);
            Console.WriteLine("Value at collect[1,0]  is " + collect[1,0]);
            Console.WriteLine("Value at collect[1,1]  is " + collect[1,1]);
            Console.WriteLine("Value at collect[2,0]  is " + collect[2,0]);
            Console.WriteLine("Value at collect[2,1]  is " + collect[2,1]);

            Console.Read();

        }
    }
}
```

With this program, the output is as follows:

Value at collect[0,0] is 1

Value at collect[0,1] is 2

Value at collect[1,0] is 3

Value at collect[1,1] is 4

Value at collect[2,0] is 5

Value at collect[2,1] is 6

It becomes easier to use loops when you iterate through multi-dimensional arrays in the same way as you would for normal arrays. Let's see how we can do this. But this time around since we have 2 dimensions we also need to use nested for loops to iterate through all the elements of the array.

Example 22: The following program is used to showcase how to iterate through a two-dimensional array.

```
using System;
```

```
namespace Demo
{
   // A simple application using C#

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

         int[,] collect = new int[3,2];

         collect[0,0] = 1;

         collect[0,1] = 2;

         collect[1,0] = 3;

         collect[1,1] = 4;

         collect[2,0] = 5;

         collect[2,1] = 6;

         // Displaying the values of the array

         for (int i = 0; i < 3; i++)
         {
            for (int j = 0; j < 2; j++)
            {
               Console.WriteLine("Value at collect["+i+","+j+"]  is " +
collect[i, j]);
            }
         }

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

Value at collect[0][0] is 1

Value at collect[0][1] is 2

Value at collect[1][0] is 3

Value at collect[1][1] is 4

Value at collect[2][0] is 5

Value at collect[2][1] is 6

4.4 Arrays as Parameters

Arrays can also be passed as parameters to functions and accessed like normal variables. Let's look at an example of how we can achieve this.

Example 23: The following program is used to showcase how to pass arrays as parameters.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      static void Display(int[] pcollect)
      {
         Console.WriteLine("The first element is " + pcollect[0]);

         Console.WriteLine("The second element is " + pcollect[1]);

         Console.WriteLine("The third element is " + pcollect[2]);
      }

      // The main function
      static void Main(string[] args)
      {

         int[] collect = new int[3];

         collect[0] = 1;

         collect[1] = 2;

         collect[2] = 3;

         //Passing the array to the method
         Display(collect);

         Console.Read();

      }
   }
```

```
}
```

With this program, the output is as follows:

The first element is 1

The second element is 2

The third element is 3

5. Numbers

We have already looked at numbers in the data types chapter in Book 1, but let's have a refresher on the different types of numbers available in C#.

Table 1: Numbers

Data type	**Description**
int	This data type is used to define a signed integer which has a range of values from -2,147,483,648 to 2,147,483,647.
unit	This data type is used to define an unsigned integer which has a range of values from 0 to 4294967295.
short	This data type is used to define a signed integer which has a range of values from -32,768 to 32,767.
long	This data type is used to define a signed long integer which has a range of values from -9223372036854775808 to 9223372036854775807.
ulong	This data type is used to define an unsigned long integer which has a range of values from 0 to 18,446,744,073,709,551,615.
float	This data type is used to define a single-precision floating point type which has a range of values from -3.402823e38 to 3.402823e38.
double	This data type is used to define a double-precision floating point type which has a

	range of values from -1.79769313486232e308 to 1.79769313486232e308.

Example 24: The following program is used to showcase the basic use of numbers in C#.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a int number
         int a = -10;
         // Defining a uint number
         uint b = 20;
         // Defining a short number
         short c = 30;
         // Defining a long number
         long d = -10000;
         // Defining a ulong number
         ulong f = 10000;
         // Defining a float number
         float g = 12.22F;
         // Defining a double number
         double h = 33.3333;

         // Displaying the numbers

         Console.WriteLine("The int value is " + a);
         Console.WriteLine("The uint value is " + b);
         Console.WriteLine("The short value is " + c);
         Console.WriteLine("The long value is " + d);
         Console.WriteLine("The ulong value is " + f);
         Console.WriteLine("The float value is " + g);
         Console.WriteLine("The double value is " + h);
```

```
            Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The int value is -10

The uint value is 20

The short value is 30

The long value is -10000

The ulong value is 10000

The float value is 12.22

The double value is 33.3333

There are a variety of functions available in C# that work with numbers. Let's look at them in more detail.

Table 2: Number Functions

Function	**Description**
Abs()	This function is used to return the absolute value of a number.
Ceiling()	This function returns the smallest integral value that is greater than or equal to the specified decimal number.
Floor()	This function returns the largest integer value that is less than or equal to the specified decimal number.
Max()	This function returns the maximum of two numbers.

Min()	This function returns the minimum of two numbers.
Pow()	This function returns a specified number raised to the specified power.
Round()	This function rounds a decimal value to the nearest integral value.
Sqrt()	This function returns the square root of a number.
Sin()	This function returns the Sine value of a particular number.
Tan()	This function returns the Tangent value of a particular number.
Cos()	This function returns the Cosine value of a particular number.

5.1 Abs Function

This function is used to return the absolute value of a number. Let's now look at an example of this function.

Example 25: The following program is used to showcase the abs function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {
         // Defining a number
         double a = 10.10;

         // Using the abs function
         Console.WriteLine("The value after applying the function is " +
Math.Abs(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 10.1

5.2 Ceiling Function

This function returns the smallest integral value that is greater than or equal to the specified decimal number. Let's now look at an example of this function.

Example 26: The following program is used to showcase the ceiling function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 10.10;

         // Using the ceiling function
         Console.WriteLine("The value after applying the function is " +
Math.Ceiling(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 11

5.3 Floor Function

This function returns the largest integer less than or equal to the specified decimal number. Let’s now look at an example of this function.

Example 27: The following program is used to showcase the floor function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 10.10;

         // Using the floor function
         Console.WriteLine("The value after applying the function is " +
Math.Floor(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 10

5.4 Max Function

This function is used to return the maximum value of two numbers. Let's now look at an example of this function.

Example 28: The following program is used to showcase the max function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 10;
         double b = 20;

         // Using the max function
         Console.WriteLine("The value after applying the function is " +
Math.Max(a,b));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 20

5.5 Min Function

This function is used to return the minimum value of two numbers. Let's now look at an example of this function.

Example 29: The following program is used to showcase the min function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 10;
         double b = 20;

         // Using the min function
         Console.WriteLine("The value after applying the function is " +
Math.Min(a,b));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 10

5.6 Pow Function

This function returns a specified number, raised to the specified power. Let's now look at an example of this function.

Example 30: The following program is used to showcase the pow function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 2;
         double b = 4;

         // Using the pow function
         Console.WriteLine("The value after applying the function is " +
Math.Pow(a,b));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 16

5.7 Round Function

This function rounds a decimal value to the nearest integral value. Let's now look at an example of this function.

Example 31: The following program is used to showcase the round function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 2.2;

         // Using the round function
         Console.WriteLine("The value after applying the function is " +
Math.Round(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 2

5.8 Sqrt Function

This function returns the square root of a number. Let's now look at an example of this function.

Example 32: The following program is used to showcase the sqrt function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 4;

         // Using the sqrt function
         Console.WriteLine("The value after applying the function is " +
Math.Sqrt(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 2

5.9 Sin Function

This function returns the sine value of a number. Let's now look at an example of this function.

Example 33: The following program is used to showcase the sin function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 45;

         // Using the sin function
         Console.WriteLine("The value after applying the function is " +
Math.Sin(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 0.85

5.10 Tan Function

This function returns the tangent value of a number. Let's now look at an example of this function.

Example 34: The following program is used to showcase the tan function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 45;

         // Using the tan function
         Console.WriteLine("The value after applying the function is " +
Math.Tan(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 1.62

5.11 Cos Function

This function returns the cosine value of a number. Let's now look at an example of this function.

Example 35: The following program is used to showcase the cos function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a number
         double a = 45;

         // Using the cos function
         Console.WriteLine("The value after applying the function is " +
Math.Cos(a));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value after applying the function is 0.53

6. Strings

A string is nothing more than a sequence of characters which are terminated by the null character to denote that the string has indeed been terminated. In C# the string is denoted by the string data type. Let's revisit how to define a string in C# via an example.

Example 36: The following program is used to showcase strings in C#.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

         // Displaying the string
         Console.WriteLine("The value of the string is " + a);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hello

There are a variety of functions available in C# to work with strings. Let's look at them in more detail.

Table 3: String Functions

Function	**Description**
length()	This function is used to return the length of the string.
ToLower()	This function returns the string in lower case.
ToUpper()	This function returns the string in upper case.
Contains()	This function is used to check the existence of another string.
Clone()	This function can be used to copy the string to another variable.
EndsWith()	This function can be used to check if the string ends with a particular string value.
IndexOf()	This is used to get the index value of a character in the string.
Insert()	This function is used to insert a string at a particular index in the source string.
Padleft()	This function is used to pad the beginning of the string.
Remove()	This function is used to remove a set of characters from a string.
Replace()	This function is used to replace a character in the original string.
Substring()	This function is used to return a substring's original string.
Trim()	This function is used to replace any trailing white spaces in the string.

PadRight()	This function is used to pad the end of the string.
TrimStart()	This function is used to trim at the start of the string.
TrimEnd()	This function is used to trim at the end of the string.
CompareTo()	This compares an instance with a specified string, and indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified string.

6.1 Length Function

This function is used to return the length of the string. Let's now look at an example of this function.

Example 37: The following program is used to showcase the length function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

         // Displaying the length string
         Console.WriteLine("The length of the string is " + a.Length);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The length of the string is 5

6.2 ToLower Function

This function returns the string in lower case. Let's now look at an example of this function.

Example 38: The following program is used to showcase the ToLower function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

          // Displaying the string in lower case
         Console.WriteLine("The lower case of the string is " +
a.ToLower());

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The lower case of the string is hello

6.3 ToUpper Function

This function returns the string in upper case. Let's now look at an example of this function.

Example 39: The following program is used to showcase the ToUpper function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

         // Displaying the string in upper case
         Console.WriteLine("The upper case of the string is " +
a.ToUpper());

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The lower case of the string is HELLO

6.4 Contains Function

This function is used to check the existence of another string. Let's now look at an example of this function.

Example 40: The following program is used to showcase the contains function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

         // Using the Contains function
         Console.WriteLine("Does the string contain the letter e " +
a.Contains("e"));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

Does the string contain the letter e True

6.5 Clone Function

This function can be used to copy the string to another variable. Let's now look at an example of this function.

Example 41: The following program is used to showcase the clone function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";
         string b;

         // Using the clone function
         b=a.Clone().ToString();
         Console.WriteLine("The value of the string is " + b);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hello

6.6 EndsWith Function

This function can be used to check if the string ends with a particular string value. Let's now look at an example of this function.

Example 42: The following program is used to showcase the EndsWith function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello World";

         // Using the EndWith function
         Console.WriteLine("Does the string end with World" +
a.EndsWith("World"));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

Does the string end with World true

6.7 IndexOf Function

This function is used to get the index of a character in a string. Let's now look at an example of this function.

Example 43: The following program is used to showcase the IndexOf function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

         // Using the IndexOf function
         Console.WriteLine("The index value of e is " + a.IndexOf('e'));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The index value of e is 1

6.8 Insert Function

This function is used to insert a string at a particular index in the source string. Let's now look at an example of this function.

Example 44: The following program is used to showcase the insert function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";
         string b;
         b=a.Insert(5, "World");

         // Displaying the new string with the inserted value
         Console.WriteLine("The value of the string is " + b);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is HelloWorld

6.9 PadLeft Function

This function is used to pad the left of the source string. Let's now look at an example of this function.

Example 45: The following program is used to showcase the PadLeft function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string str = "pad-left";
         char pad = '.';

         Console.WriteLine(str.PadLeft(10, pad));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

..pad-left

6.10 Remove Function

This function is used to remove a set of characters from a string. Let's now look at an example of this function.

Example 46: The following program is used to showcase the remove function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "HelloWorld";
         string b;
         b=a.Remove(0, 5);

         Console.WriteLine("The value of the string is " + b);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is World

6.11 Replace Function

This function is used to replace a character in the original string. Let's now look at an example of this function.

Example 47: The following program is used to showcase the replace function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";
         string b;
         b = a.Replace('e', 'a');

         Console.WriteLine("The value of the string is " + b);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hallo

6.12 Substring Function

This function is used to return a substring's original string. Let's now look at an example of this function.

Example 48: The following program is used to showcase the substring function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "HelloWorld";
         string b;
         b = a.Substring(0, 5);

         Console.WriteLine("The value of the string is " + b);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hello

6.13 Trim Function

This function is used to trim all whitespaces in a string. Let's now look at an example of this function.

Example 49: The following program is used to showcase the trim function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello  ";
         string b;
         b = a.Trim();

         Console.WriteLine("The value of the string is " + b);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hello

6.14 PadRight Function

This function is used to pad the end of the source string. Let's now look at an example of this function.

Example 50: The following program is used to showcase the PadRight function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

         Console.WriteLine("The value of the string is " +
a.PadRight(10,'.'));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hello.....

6.15 TrimStart Function

This function is used to trim the beginning of a string. Let's now look at an example of this function.

Example 51: The following program is used to showcase the TrimStart function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "  Hello";

         Console.WriteLine("The value of the string is " +
a.TrimStart());

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hello

6.16 TrimEnd Function

This function is used to trim the end of a string. Let's now look at an example of this function.

Example 52: The following program is used to showcase the TrimEnd function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello   ";

         Console.WriteLine("The value of the string is " +
a.TrimEnd());

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The value of the string is Hello

6.17 CompareTo Function

This compares an instance with a specified string, and indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified string.

- If the value is less than zero it means that the destination string precedes the value.
- If the value is greater than zero it means that the destination string follows the value.
- If the value is zero then this instance has the same position in the sort order as the value.

Let's now look at an example of this function.

Example 53: The following program is used to showcase the CompareTo function.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {

      // The main function
      static void Main(string[] args)
      {

         // Defining a string
         string a = "Hello";

         Console.WriteLine("The output is " + a.CompareTo("Hello"));

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The output is 0

7. Structures in C#

The data structure type is used to combine data items into a logical unit. Each data item can be of a different type. Structures are used to store records, which was the commonly used method of storing data in the past. The syntax of the structure data type is shown below.

```
struct structurename
{
Datatype membername1;
Datatype membername2;
..
Datatype membernameN;
};
```

Where:

- structurename is the name of the structure.
- membername is the name of the various members in the data structure.

The declaration of a structure in a C# program would look something like this:

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      struct Student
      {
         public int studentID;
         public string studentName;
      };
      // The main function
      static void Main(string[] args)
      {

         Console.Read();
```

```
        }
    }
}
```

7.1 Accessing the Members of the Structure

The members of the structure can be accessed via the dot operator. Let's look at an example of how this can be achieved.

Example 54: The following program is used to showcase how to access members of a structure.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      struct Student
      {
         public int studentID;
         public string studentName;
      };
      // The main function
      static void Main(string[] args)
      {

         Student stud;
         // Setting the values of the data structure
         stud.studentID = 1;
         stud.studentName = "Mark";

         // Displaying the values of the data structure
         Console.WriteLine("The ID of the student is " +
stud.studentID);
         Console.WriteLine("The name of the student is " +
stud.studentName);

         Console.Read();

      }
   }
}
```

With this program, the output is as follows:

The ID of the student is 1

The ID of the student is Mark

One can also define multiple variables of the structure. Let's look at an example of this.

Example 55: The following program is used to showcase how to define multiple variables of a data structure.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      struct Student
      {
         public int studentID;
         public string studentName;
      };
      // The main function
      static void Main(string[] args)
      {

         Student stud1,stud2;

         // Setting the values of the data structure
         stud1.studentID = 1;
         stud1.studentName = "Mark";

         // Displaying the values of the data structure
         Console.WriteLine("The ID of the student is " +
stud1.studentID);
         Console.WriteLine("The name of the student is " +
stud1.studentName);

         // Setting the values of the data structure
         stud2.studentID = 2;
         stud2.studentName = "John";

         // Displaying the values of the data structure
         Console.WriteLine("The ID of the student is " +
```

```
stud2.studentID);
            Console.WriteLine("The name of the student is " +
stud2.studentName);

            Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The ID of the student is 1

The ID of the student is Mark

The ID of the student is 2

The ID of the student is John

7.2 Passing Structures as Parameters

Lastly, structures can also be passed as parameters to functions. Let's look at an example of how we can pass structures as parameters.

Example 56: The following program is used to showcase how to use data structures as parameters.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      struct Student
      {
         public int studentID;
         public string studentName;
      };

      static void Display(Student pStudent)
      {

         // Displaying the values of the data structure
         Console.WriteLine("The ID of the student is " +
pStudent.studentID);
         Console.WriteLine("The name of the student is " +
pStudent.studentName);

      }
      // The main function
      static void Main(string[] args)
      {

         Student stud1,stud2;

         // Setting the values of the data structure
         stud1.studentID = 1;
         stud1.studentName = "Mark";

         // Displaying the values of the data structure
         Display(stud1);
```

```
            // Setting the values of the data structure
            stud2.studentID = 2;
            stud2.studentName = "John";

            // Displaying the values of the data structure
            Display(stud2);

            Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The ID of the student is 1

The ID of the student is Mark

The ID of the student is 2

The ID of the student is John

8. Enum Data Type

The enum data type is used to define a custom data type. So if you need to define custom values, an enum type is your go-to. This is best explained through an example.

Below, we are required to define a type which can store the days of the week. Since the days of the week cannot be stored in the data types we have already seen, we need to define this using the enum type.

Example 57: The following program is used to showcase how to use enum types.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      // Declaring the enum data type
      enum Daysoftheweek
      {
         Sunday, // assigned a value of 0
         Monday, // assigned a value of 1
         Tuesday, // assigned a value of 2
         Wednesday, // assigned a value of 3
         Thursday, // assigned a value of 4
         Friday, // assigned a value of 5
         Saturday // assigned a value of 6
      };

      // The main function
      static void Main(string[] args)
      {

         Daysoftheweek holiday = Daysoftheweek.Sunday ;

         Console.WriteLine("The day of the week is " + holiday);

         Console.Read();
```

```
        }
    }
}
```

Some of the key aspects which need to be noted about the above program are:

- All the values for the enum type collection are defined in the curly braces.
- Internally, the first value of the collection will be assigned an integer value of 0.
- The enum type needs to end with a semi colon.
- One can define variables of the enum type.

With this program, the output is as follows:

The day of the week is Sunday

Now let's look at another example of the enum type. This time we are going to explicitly define the value for the first item in the enum collection.

Example 58: The following program is used to define the starting value of the first item in the enum collection.

```
using System;

namespace Demo
{
    // A simple application using C#

    class Program
    {
        // Declaring the enum data type
        enum Daysoftheweek
        {
            Sunday=-3, // assigned a value of -3
            Monday, // assigned a value of -2
            Tuesday, // assigned a value of -1
```

```
        Wednesday, // assigned a value of 0
        Thursday, // assigned a value of 1
        Friday, // assigned a value of 2
        Saturday // assigned a value of 3
    };

    // The main function
    static void Main(string[] args)
    {

        Daysoftheweek holiday = Daysoftheweek.Friday ;

        Console.WriteLine("The day of the week is " + holiday);

        Console.Read();

    }
  }
}
```

In the above program you can see that we are explicitly defining the value of the first item in the collection to the value of -3. The rest of the items in the collection will get the subsequent integer values.

With this program, the output is as follows:

The day of the week is Friday

Now let's look at another example wherein we can loop through the values of the enum type if we don't want to show the integer value but the actual item in the enum collection.

Example 59: The following program shows how to use conditions to see the value corresponding to the item in the enum collection.

```
using System;

namespace Demo
{
  // A simple application using C#

  class Program
  {
```

```
        // Declaring the enum data type
        enum Daysoftheweek
        {
            Sunday=-3, // assigned a value of -3
            Monday, // assigned a value of -2
            Tuesday, // assigned a value of -1
            Wednesday, // assigned a value of 0
            Thursday, // assigned a value of 1
            Friday, // assigned a value of 2
            Saturday // assigned a value of 3
        };

        // The main function
        static void Main(string[] args)
        {

            Daysoftheweek holiday = Daysoftheweek.Friday ;

    if (holiday == Daysoftheweek.Sunday)
Console.WriteLine("Sunday");
    else if (holiday == Daysoftheweek.Monday)
Console.WriteLine("Monday");
    else if (holiday == Daysoftheweek.Tuesday)
Console.WriteLine("Tuesday");
    else if (holiday == Daysoftheweek.Wednesday)
Console.WriteLine("Wednesday");
    else if (holiday == Daysoftheweek.Thursday)
Console.WriteLine("Thursday");
    else if (holiday == Daysoftheweek.Friday)
Console.WriteLine("Friday");
    else Console.WriteLine("Saturday");

            Console.Read();

        }
    }
}
```

With this program, the output is as follows:

Friday

9. Classes and Objects

A class is used to represent entities. For example, suppose we wanted to represent a student using C#, we can do this with the help of a class. The student will be in the form of a class and will have something known as properties. These properties can be StudentID, StudentName or anything else that can be used to define the student.

The class can also have methods, which can be used to manipulate the properties of the student. Therefore there can be a method that can be used to define the values of the properties and another to display the properties of the student. The syntax for the definition of a class is:

```
Class classname
{
Class modifier Data type Class members;
Class modifier Data type Class functions;
}
```

Where:

- classname is the name given to the class.
- The class modifier is the visibility given to either the member or function of a class, which can be private, public or protected.
- We then define the various data members of a class, each of which can have a data type.
- We then have the functions of the class, each can accept parameters and also return values of different data types.

So let's look at a quick sample of a "student" class.

```
class Student
{
int studentID;
string studentName;
}
```

So in the above code we have:

- A class with the name of Student.
- Two properties. One is studentID, which is an integer type and the other is studentName, which is a string type.

In order to define a student with values, we need to define an object of the class. To define an object, we just need to ensure we define the type as that of the class.

```
Student student1;
```

In the above code, student1 is the variable which is of the type Student. We have to now define values to the properties. Let's look at an example of how to define classes and objects.

Example 60: The following program is used to showcase how to use classes and objects.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
   class Student
{
   // The members of the class
   public int studentID;
   public string studentName;
};

       // The main function
       static void Main(string[] args)

       {
       // variable of the type student
       Student stud1=new Student();
```

```
        // Setting the values of the Student object
        stud1.studentID=1;
        stud1.studentName="John";

            // Displaying the values of the Student object
            Console.WriteLine("The ID of the student is " +
stud1.studentID);
            Console.WriteLine("The name of the student is " +
stud1.studentName);
            Console.Read();
          }
    }
}
```

Now with the above program:

- We have defined a class called Student outside of the main program.
- The Student class has two properties. They both have the public modifier (we will look at modifiers shortly).
- We then defined an object called stud1 of the type Student.
- We then assigned a value of 1 to studentID and John to studentName.
- We then displayed the values to the console.

With this program, the output is as follows:

The ID of the student is 1

The name of the student is John

9.1 Member Functions

Member functions can be used to add more functionality to classes. A member function can be used, for example, to output the values of properties. This means that you wouldn't need to add code every time you wanted to display the values, you could just invoke the function. Let's look at an example of how we can use member functions.

Example 61: The following program is used to showcase how to use member functions.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
   class Student

{
   // The members of the class
   public int studentID;
   public string studentName;
   // Declaring a member function
   public void Display()

   {
   Console.WriteLine("The ID of the student is " + studentID);
   Console.WriteLine("The name of the student is " + studentName);
   }
};

      // The main function
      static void Main(string[] args)
      {
      // variable of the type student
      Student stud1=new Student();

      // Setting the values of the Student object
      stud1.studentID=1;
      stud1.studentName="John";
```

```
        // Displaying the values of the Student object
        stud1.Display();
        Console.Read();
        }
    }
}
```

With the above program:

- We are now defining a member function called Display() which outputs the studentID and studentName to the console.
- We can then call the member function from the object in the main program.

With this program, the output is as follows:

The ID of the student is 1

The name of the student is John

Now let's look at a way we can use member functions to take values from the main program.

Example 62: The following program is used to showcase how to use member functions in another way.

```
using System;

namespace Demo
{
    // A simple application using C#

    class Program
    {

    class Student
{
    // The members of the class
    public int studentID;
    public string studentName;
    // Declaring a member function
```

```
    public void Display()

    {
    Console.WriteLine("The ID of the student is " + studentID);
    Console.WriteLine("The name of the student is " + studentName);
    }

    public void Input(int id, string name)
    {
    studentID = id;
    studentName = name;
    }
};

        // The main function
        static void Main(string[] args)
        {
        // variable of the type student
        Student stud1=new Student();

        // Setting the values of the Student object
        stud1.Input(1,"John");

        // Displaying the values of the Student object
        stud1.Display();
        Console.Read();

        }
    }
}
```

With the above program we are:

- Defining a member function called Input() that takes in two values and assigns them to the studentID and studentName property.

With this program, the output is as follows:

The ID of the student is 1

The name of the student is John

9.2 Class Modifiers

Class modifiers can be used to define the visibility of properties and methods in a class. Below are the various modifiers available.

1. Private: The properties and methods are only available to the class itself.
2. Protected: The properties and methods are only available to the class itself and subclasses derived from that class.
3. Public: The properties and methods are available to all classes.

Let's look at an example for using a private class modifier.

Example 63: The following program is used to showcase how to use private access modifiers with an error condition.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
   class Student
{
   // The members of the class
   private int studentID;
   public string studentName;
   // Declaring a member function
   public void Display()

   {
   Console.WriteLine("The ID of the student is " + studentID);
   Console.WriteLine("The name of the student is " + studentName);
    }
};

       // The main function
       static void Main(string[] args)
       {
       // variable of the type student
```

```
        Student stud1=new Student();

        // Setting the values of the Student object
        stud1.StudentID = 1;
        stud1.studentName = "John";

        // Displaying the values of the Student object
        stud1.Display();
        Console.Read();
        }
    }
}
```

With this program, the output is as follows:

Error 1 'Demo.Program.Student' does not contain a definition for 'StudentID' and no extension method 'StudentID' accepting a first argument of type 'Demo.Program.Student' could be found (are you missing a using directive or an assembly reference?) G:\Code\Demo\Demo\Program.cs 37 11 Demo

Since the access modifier is private, it can only be accessed by the class itself. Hence you will get the above errors. The way to resolve this is to define the program as follows.

Example 64: The following program is used to showcase how to use private access modifiers.

```
using System;

namespace Demo
{
    // A simple application using C#

    class Program
    {
    class Student
{
    // The members of the class
    private int studentID;
    public string studentName;
    // Declaring a member function
```

```
    public void Display()

    {
    Console.WriteLine("The ID of the student is " + studentID);
    Console.WriteLine("The name of the student is " + studentName);
    }

    public void Input(int id, string name)
    {
    studentID = id;
    studentName = name;
    }
};

        // The main function
        static void Main(string[] args)
        {
        // variable of the type student
        Student stud1=new Student();

        // Setting the values of the Student object
        stud1.Input(1,"John");

        // Displaying the values of the Student object
        stud1.Display();
        Console.Read();
        }
    }
}
```

Now with the above program:

- We are defining the properties as private and the methods as public. This is the normal practice for classes.

With this program, the output is as follows:

Student ID 1

Student Name John

One can also define multiple classes in a program and make use of those classes. Let’s look at an example of this.

Example 65: The following program is used to showcase how to use multiple classes.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
   class Student

{
   // The members of the class
   private int studentID;
   public string studentName;
   // Declaring a member function
   public void Display()

   {
   Console.WriteLine("The ID of the student is " + studentID);
   Console.WriteLine("The name of the student is " + studentName);
   }
   public void Input(int id, string name)
   {
   studentID = id;
   studentName = name;
   }
};

       class Employee
       {
          // The members of the class
          private int EmployeeID;
          public string EmployeeName;
          // Declaring a member function
          public void Display()
          {

              Console.WriteLine("The ID of the Employee is " +
EmployeeID);
              Console.WriteLine("The name of the Employee is " +
EmployeeName);

          }
          public void Input(int id, string name)
```

```
        {
            EmployeeID = id;
            EmployeeName = name;
        }
    };

    // The main function
    static void Main(string[] args)
    {
    // variable of the type student
    Student stud1=new Student();

    // Setting the values of the Student object
    stud1.Input(1, "John");
    // Displaying the values of the Student object
    stud1.Display();

    // variable of the type Employee
    Employee emp1 = new Employee();

    // Setting the values of the Employee object
    emp1.Input(2, "Mark");
    // Displaying the values of the Employee object
    emp1.Display();

    Console.Read();

    }
  }
}
```

With the above program:

- We are defining two classes; one is Employee and the other is Student. We can use both of these classes in the main program.

With this program, the output is as follows:

The ID of the student is 1

The name of the student is John

The ID of the Employee is 2

The name of the Employee is Mark

We can also define multiple objects of a class. Let's look at an example of this.

Example 66: The following program is used to showcase how to define multiple objects.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      class Student
{
   // The members of the class
   private int studentID;
   public string studentName;

   // Declaring a member function
   public void Display()

   {
      Console.WriteLine("The ID of the student is " + studentID);
      Console.WriteLine("The name of the student is " + studentName);
   }

   public void Input(int id, string name)
   {
      studentID = id;
      studentName = name;
   }
};

      // The main function
      static void Main(string[] args)
      {

   // variable of the type student
   Student stud1=new Student();
   Student stud2= new Student();

   // Setting the values of the Student object
```

```
    stud1.Input(1, "John");
    stud2.Input(2, "Mark");

    // Displaying the values of the Student object
    stud1.Display();
    stud2.Display();

        Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The ID of the student is 1

The name of the student is John

The ID of the Employee is 2

The name of the Employee is Mark

9.3 Constructors

Constructors are special methods in a class which are called when an object of the class is created. The constructor has the name of class. The syntax of the constructor method is shown below.

```
public classname()
{
// Define any code for the constructor
}
```

Let's look at an example of how we can use constructors.

Example 67: The following program is used to showcase how to define constructors.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
        class Student
{
   // The members of the class
   private int studentID;
   public string studentName;

   // Declaring a member function
   public void Display()

   {
      Console.WriteLine("The ID of the student is " + studentID);
      Console.WriteLine("The name of the student is " + studentName);
   }
   public Student()

   {
      Console.WriteLine("The constructor is being called ");
      studentID = 1;
      studentName = "Default";
```

```
    }

    public void Input(int id, string name)

    {
        studentID = id;
        studentName = name;
    }
};

        // The main function
        static void Main(string[] args)

        {
    // variable of the type student
    Student stud1=new Student();
    stud1.Display();

    // Setting the values of the Student object
    stud1.Input(1, "John");

    // Displaying the values of the Student object
    stud1.Display();

        Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The constructor is being called

The ID of the student is 1

The name of the student is Default

The ID of the student is 1

The name of the student is John

As you can see from the output, the constructor gets called when the object is created and assigns the default values to the properties of the class.

9.4 Parameterized Constructors

It is also possible to pass parameters to the constructor as you would with any other ordinary function. The syntax of the constructor method with parameters is shown below.

```
classname(parameters)
{
// Use the parameters accordingly.
}
```

Let's look at an example on how we can use parameterized constructors.

Example 68: The following program is used to showcase how to work with parameterized constructors.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      class Student
{
   // The members of the class
   private int studentID;
   public string studentName;

   // Declaring a member function
   public void Display()

   {
      Console.WriteLine("The ID of the student is " + studentID);
      Console.WriteLine("The name of the student is " + studentName);
      }

   public Student(int id, string name)
   {
      Console.WriteLine("The constructor is being called ");
      studentID = id;
```

```
        studentName = name;
    }
};

        // The main function
        static void Main(string[] args)
        {
        // variable of the type student

        // Setting the values of the Student object
        Student stud1=new Student(1,"John");

        // Displaying the values of the Student object
        stud1.Display();

        Console.Read();

         }
    }
}
```

Now with the above program we are:

- Defining a constructor which takes in parameters.
- We are then assigning the parameters to the properties in the constructor.

With this program, the output is as follows:

The constructor is being called

The ID of the student is 1

The name of the student is John

9.5 Destructors

Destructors are special methods in a class which are called when an object of the class is destroyed. The constructor has the name of class along with the ~ character. The syntax of the destructor method is shown below.

```
~classname
{
}
```

Let’s look at an example of how we can use destructors.

Example 69: The following program is used to showcase how to work with destructors.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Program
   {
      class Student
{
   // The members of the class
   private int studentID;
   public string studentName;

   // Declaring a member function
   public void Display()

   {
      Console.WriteLine("The ID of the student is " + studentID);
      Console.WriteLine("The name of the student is " + studentName);
      }

   public Student(int id, string name)
   {
      Console.WriteLine("The constructor is being called ");
      studentID = id;

      studentName = name;
```

```
    }

    ~Student()
    {
        Console.WriteLine("The destructor is being called ");
    }
};

        // The main function
        static void Main(string[] args)
        {
        // variable of the type student
        // Setting the values of the Student object
        Student stud1=new Student(1,"John");

        // Displaying the values of the Student object
        stud1.Display();

        Console.Read();

        }
    }
}
```

With the above program we are:

- Defining a destructor function which has the class name along with the ~ character.

9.6 Static Members

Static members are used to define values that should remain the same across all objects of a class. Since classes have properties which are separate for each object, sometimes there is a requirement to have properties that never change. This can be done with static members. The syntax of static member definitions is shown below.

```
Class classname
{
static datatype staticmembername
}
```

Here the static keyword is used to show that a static member is being defined. If you want to initialize the static data member, this can be done outside of the class as follows.

```
datatype classname::staticmembername = value;
```

Let's look at an example of how we can achieve this.

Example 70: The following program is used to showcase how to work with static members.

```
using System;

namespace Demo
{
   // A simple application using C#
   class Program
   {
      class Student
{

   // The members of the class
   public static int Counter=1;
   private int studentID;
   public string studentName;

   // Declaring a member function
```

```
    public void Display()

    {
        Console.WriteLine("The ID of the student is " + studentID);
        Console.WriteLine("The name of the student is " + studentName);
    }

    public Student(int id, string name)
    {
        Counter++;
        studentID = id;

        studentName = name;
    }
};

        // The main function
        static void Main(string[] args)
        {

        // variable of the type student
        // Setting the values of the Student object
        Student stud1=new Student(1,"John");

        // Displaying the values of the Student object
        Console.WriteLine("The value of the counter is " +
Student.Counter);

        Student stud2 = new Student(1, "John");

        Console.WriteLine("The value of the counter is " +
Student.Counter);

        Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The value of the counter is 1

The value of the counter is 2

10. Inheritance

We have looked at classes and objects in the previous chapter, and have seen how to define classes with properties and functions. Before we jump into class inheritance lets quickly revisit how a typical class looks like.

Example 71: The following program is used to showcase how to use a simple class.

```
using System;

namespace Demo
{
   // A simple application using C#

   class Student
   {
      public int studentID;
      public string studentName;

      // Declaring a member function
      public void Display()

      {
         Console.WriteLine("The ID of the student is " + studentID);
         Console.WriteLine("The name of the student is " +
studentName);
      }
   };

   class Program
   {
      // The main function
      static void Main(string[] args)

      {
      // variable of the type student
      // Setting the values of the Student object
      Student stud1=new Student();
      stud1.studentID = 1;
      stud1.studentName = "John";
```

```
        // Displaying the values of the Student object
        stud1.Display();

        Console.Read();

        }
    }
}
```

With this program:

- We have a class called Student which has 2 members, one is StudentID and the other is StudentName.
- We then define a member function called Display() which outputs the StudentID and StudentName to the console.
- We can then call the member function from the object in the main program.

With this program, the output is as follows:

The ID of the student is 1

The Name of the student is John

10.1 What is Inheritance?

Inheritance is a concept wherein we can define a class to inherit the properties and methods of another class. This helps in not having the need to define the class again and having the properties and methods defined again.

Let's say that we had a class called Person, which had a property of Name and a method of Display. Then via inheritance, we can define a class called Student which could inherit the Person class. The Student class would automatically get the Name member and the Display function. The Student class could then define its own additional members if required. To define an inherited class we use the following syntax.

```
Derived class:Base class
```

Here the Derived class is the class which will inherit the properties of the other class, known as the base class. If we had a base class with a property and a function as shown below:

```
Base class
{
Public or protected Property1;
}
```

Then when we define the derived class from the base class, the derived class will have access to the property. Note that the property and function needs to have the access modifier as public or protected.

```
Derived class: Base Class
{
// No need to define property1, it will automatically inherit these.
}
```

Now let's look at a simple example of inheritance via code.

Example 72: The following program is used to showcase how to use a

simple inherited class.

```
using System;

namespace Demo
{
   // A simple application using C#

   // Defining the base class
   class Person
   {
      public string Name;
   }

   class Student:Person
   {
      public int ID;

      // Declaring a member function
      public void Display()

      {
         Console.WriteLine("The ID of the student is " + ID);
         Console.WriteLine("The name of the student is " + Name);
      }
   };

   class Program
   {
      // The main function
      static void Main(string[] args)
      {

      // variable of the type student
      // Setting the values of the Student object
      Student stud1=new Student();
      stud1.ID = 1;
      stud1.Name = "John";

      // Displaying the values of the Student object
      stud1.Display();

      Console.Read();

      }
   }
}
```

Now with the above program:

- We are defining a class called Person which has one member called Name.
- We then use inheritance to define the Student class. Notice that we now define another property called ID
- In the Display function, note that we can use the Name property without the need of defining it in the Student class again.

With this program, the output is as follows:

The ID of the student is 1

The Name of the student is John

So now we have seen how we can use derived and base classes and this is known as inheritance.

10.2 Functions in Derived Classes

We can also define functions which can be inherited from the base class. Let's see how we can achieve this. If we have a base class with a property and a function as shown below.

```
Base class
{
Public or protected Property1;
Public or protected Fucntion1;
}
```

Then when we define the derived class from the base class, the derived class will have access to the property and the function as well. Note that the property and function need to either have the access modifier as public or protected.

```
Derived class: Base Class
{
// No need to define property1 and Function1, it will automatically
inherit these.
}
```

Let's look at an example where we can use functions in derived classes.

Example 73: The following program is used to showcase how to use an inherited class with functions.

```
using System;

namespace Demo
{
   // A simple application using C#

   // Defining the base class
   class Person
   {
      public string Name;
      public int ID;

      // Declaring a member function
```

```
        public void Display()

        {
            Console.WriteLine("The ID of the student is " + ID);
            Console.WriteLine("The name of the student is " + Name);
        }
    }
    class Student:Person
    {
    };

    class Program
    {
         // The main function
        static void Main(string[] args)
        {
        // variable of the type student
        // Setting the values of the Student object
        Student stud1=new Student();
        stud1.ID = 1;
        stud1.Name = "John";

        // Displaying the values of the Student object
        stud1.Display();

        Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The ID of the student is 1

The Name of the student is John

You can also redefine the Display function in the Student class. We will see more of this in the operating overloading chapter.

If you look at the above example, you will notice that the Display function in the Person class has the display text as ID and Name. But suppose we wanted to have the display name as Student ID and Student Name in the student class, we can redefine the Display function.

In order to ensure the derived class overrides the base class functions, we have to use the “new” keyword in the derived class function. Let’s look at an example of this.

Example 74: The following program is used to showcase how to use an inherited class with redefined functions.

```
using System;

namespace Demo
{
   // A simple application using C#

   // Defining the base class
   class Person
   {
      public string Name;
      public int ID;

      // Declaring a member function
      public void Display()

      {
         Console.WriteLine("The ID is " + ID);
         Console.WriteLine("The name " + Name);
      }
   }

   class Student:Person
   {
      public new void Display()
      {

         Console.WriteLine("The ID of the student is " + ID);
         Console.WriteLine("The name of the student is " + Name);
      }
   };

   class Program
   {
      // The main function
      static void Main(string[] args)
      {
```

```
        // variable of the type student
        // Setting the values of the Student object
        Student stud1=new Student();
        stud1.ID = 1;
        stud1.Name = "John";

        // Displaying the values of the Student object
        stud1.Display();

        Console.Read();

        }
    }
}
```

With the above program:

- Note that we are using the “new” keyword for the Display function in the Student class. Only then will the derived class Display be called.

With this program, the output is as follows:

The ID of the student is 1

The name of the student is John

10.3 Access Control

In the earlier chapters, we have seen the access modifiers named private, public and protected. Let's again revisit the definition of these modifiers. Class modifiers can be used to define the visibility of properties and methods in a class. Below are the various modifiers available.

1. Private – The properties and methods are only available to the class itself.
2. Protected - The properties and methods are only available to the class itself and subclasses derived from that class.
3. Public - The properties and methods are available to all classes.

Let's say we have the following structure for a class.

```
Class classname
{
Private Property1;
Private Fucntion1;
}
```

Both property1 and function1 would not be accessible anywhere else except from the class it is defined in. If you try to access any of these properties or functions from the main program, you would get a compile time error. Now let's define a class again but this time use the protected access modifier.

```
Class classname1
{
Protected Property1;
Protected Fucntion1;
}
```

Now property1 and function1 are accessible from both the above class and any other class which derives from this class. So if we had the derived class definition below, we would be able to access both property1 and fucntion1 in the derived class. But note that we could not use these properties in any other non-derived class.

```
Class derivedclassname: classname1
{
}
```

Let’ look at an example on using access modifiers.

Example 75: The following program is used to showcase how to use access modifiers.

```
using System;

namespace Demo
{
   // A simple application using C#

   // Defining the base class
   class Person
   {
      protected string Name;
      public int ID;
      private string city;

      // Declaring a member function
      public void Display()

      {
         Console.WriteLine("The ID is " + ID);
         Console.WriteLine("The name " + Name);
      }
   }

   class Student:Person
   {
      public new void Display()
      {
         Console.WriteLine("The ID of the student is " + ID);
         Console.WriteLine("The name of the student is " + Name);
         Console.WriteLine("The city of the student is " + city);
      }
   };

   class Program
   {
```

```
        // The main function
        static void Main(string[] args)
        {
        // variable of the type student
        // Setting the values of the Student object
        Student stud1=new Student();
        stud1.ID = 1;
        stud1.Name = "John";

        // Displaying the values of the Student object
        stud1.Display();

        Console.Read();

        }
    }
}
```

With this program, the output is as follows:

Error 2 'Demo.Person.city' is inaccessible due to its protection level G:\Code\Demo\Demo\Program.cs 31 63 Demo

Error 3 'Demo.Person.Name' is inaccessible due to its protection level G:\Code\Demo\Demo\Program.cs 46 15 Demo

So why are we getting this error? Let's inspect the different parts of the program.

- The person class has 2 properties called ID and Name. Name has the protected access modifier, which means that it can only be accessed in the derived class. However we are trying to access it in the main class which is giving the error.
- Secondly we are trying to access the city attribute, which has been marked as private in the derived class. This is the reason for the second error.

The correct way to implement the above program is as follows.

Example 76: The following program is used to showcase how to use

access modifiers in the proper way.

```
using System;

namespace Demo
{
   // A simple application using C#

   // Defining the base class
   class Person
   {
      protected string Name;
      public int ID;

      // Declaring a member function
      public void Display()

      {
         Console.WriteLine("The ID is " + ID);
         Console.WriteLine("The name " + Name);
      }
   }

   class Student:Person
   {
      public void Input(string pName)
      {
         Name = pName;
      }
      public new void Display()

      {
         Console.WriteLine("The ID of the student is " + ID);
         Console.WriteLine("The name of the student is " + Name);
      }
   };

   class Program
   {

      // The main function
      static void Main(string[] args)
      {
      // variable of the type student
      // Setting the values of the Student object
      Student stud1=new Student();
      stud1.ID = 1;
```

```
        stud1.Input("John");

        // Displaying the values of the Student object
        stud1.Display();

        Console.Read();

        }
    }
}
```

In this program we now correctly define a method called InputName in the Student class, that can access the protected Name member.

With this program, the output is as follows:

The ID of the student is 1

The Name of the student is John

11. Polymorphism

Polymorphism refers to classes in which base and derived classes can have the same functions with the same names. But which option gets called, depends on the type of class from which the function gets called. So let's say that we have a base class as defined below.

```
class baseclassA
{
FunctionA() { }
}
```

And then we have the derived class as shown below.

```
Class derivedclass :  baseclassA
{
FunctionA() { }
}
```

Additionally in the main program we define the object as:

```
derivedclass objA;
```

Then we call FunctionA. This would actually call the FunctionA which is defined in the derived class and not the one in the base class.

```
objA.FucntionA();
```

Let's look at an example to understand this better.

Example 77: The following program is used to showcase a simple example of polymorphism.

```
using System;

namespace Demo
{
    class Person
```

```
{
        public int ID;
        public string Name;

    public void Display()
    {
        Console.WriteLine("The ID is " +ID);
        Console.WriteLine("The Name is " +Name);
    }
    public void InputName(string pName)
    {
        Name=pName;
    }
};

class Student:Person {
    public new void InputName(string pName)
    {
        Name=pName;
    }
    public new void Display()

    {
        Console.WriteLine("The Student ID is " + ID);
        Console.WriteLine("The Student Name is " + Name);
    }
};

    class Program
    {
        // The main function
        static void Main(string[] args)
        {
            // variable of the type student
            // Setting the values of the Student object
            Student stud1=new Student();

            stud1.ID = 1;
            stud1.InputName("John");
            stud1.Display();
            Person per1=new Person();
            per1.ID = 2;
            per1.InputName("Mark");
            per1.Display();

        Console.Read();
```

```
        }
    }
}
```

With the above program:

- We now have 2 separate classes; one is Person and the other is the Student derived class.
- We then have 2 objects, each defined of each class. When we call the InputName and Display functions, we can see that the function gets called depending on which class the variable is defined as.

With this program, the output is as follows:

The Student ID is 1

The Student Name is John

The ID is 2

The Name is Mark

12. Operator Overloading

Operator overloading is the concept of adding custom functionality to an existing operator available in C#.

So why would you want to define the additional operator? It's used to expand the functions of what the current operator can do. Let's take the addition operator. We know that we can use this operator for mathematical additions. But suppose we want to add multiple values of a class. If we try to just add the objects together, it wouldn't work. So let's say we had a class definition as follows.

```
class classA
{
}
```

And then in the main program we create 2 objects from that class.

```
classA objA,objB
```

If we try to add the objects, what would we get?

```
objA+objB
```

Well both objA and objB don't contain any values of the properties in the object, but rather they contain the memory addresses assigned to the object. So in the end you would just be adding two memory addresses, and that's not what we want. If we have the class definition as follows.

```
class classA
{
int i;
}
```

And then create two objects from that class and assign a value to i:

```
classA objA,objB
objA.i=4;
```

```
objB.i=5;
```

Now when we perform the operation objA+objB we want the output to be 9. We want the property 'i' to be added from both objects. In these situations where we want to extend the functionality of an operator, we would make use of operator overloading. The general syntax for operator overloading is shown below.

```
operator operatortobeoverloaded()
{
//Add your code here
}
```

The following table shows the operators that can be overloaded.

Table 4: Operators That Can Be Overloaded

Operators	**Description**
int	This data type is used to define a signed integer which has a range of values from -2,147,483,648 to 2,147,483,647.
+, -, !, ~, ++, --	These unary operators take one operand and can be overloaded.
+, -, *, /, %	These binary operators take one operand and can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded as well.
&&, ||	The conditional logical operators cannot be

	overloaded directly.
+=, -=, *=, /=, %=	The assignment operators cannot be overloaded.
=, ., ?:, ->, new, is, sizeof, typeof	These operators cannot be overloaded either.

It’s best to see this in practice through an example.

Example 78: The following program is used to showcase how to use operator overloading.

```
using System;

namespace Demo
{
   class Rectangle {
      public int height;
      public int length;
      public int Volume;

public void Display()
{
   Console.WriteLine("The volume is "+Volume);
}

   // Overload + operator to add two rectangle objects.
public static Rectangle operator +(Rectangle aRect, Rectangle bRect)
{
   Rectangle pRect = new Rectangle();
   pRect.length = aRect.length + bRect.length;
   pRect.height = aRect.height + bRect.height;
   pRect.Volume = pRect.length * pRect.height;
   return pRect;
}
   };

   class Program
   {

      // The main function
      static void Main(string[] args)
```

```
    {
        // variable of the type student
        // Setting the values of the Student object

        Rectangle Rect1 = new Rectangle();
        Rectangle Rect2 = new Rectangle();
        Rectangle Rect3 = new Rectangle();

        Rect1.length = 2;
        Rect1.height = 2;

        Rect2.length = 3;
        Rect2.height = 4;

        Rect3 = Rect1 + Rect2;
        Rect3.Display();

    Console.Read();

    }
  }
}
```

With the above program:

- We define a class called Rectangle which has three properties: length, height and volume.
- Next we want to apply the + operator to the class objects to be able to add the properties of the members.

We can't just add the objects together, because that would add the memory addresses assigned to the objects. We need to redefine the + operator for this, and we do this with the following definition.

```
public static Rectangle operator +(Rectangle aRect, Rectangle bRect)
```

In this function we combine the volume combination of the rectangles and assign it to the combined rectangle. We then return it to the calling program. We now have the ability to add the properties of both the rectangles via the + operator.

With this program, the output is as follows:

The volume is 30.

Example 79: The following program is used to showcase another example of how to use operator overloading.

```
using System;

namespace Demo
{
   class Rectangle {
      public int height;
      public int length;
      public int Volume;

public void Display()
{
   Console.WriteLine("The volume is "+Volume);
}

   // Overload + operator to add two rectangle objects.
public static Boolean operator <(Rectangle aRect, Rectangle bRect)
{
   Rectangle pRect = new Rectangle();
   aRect.Volume = aRect.length * aRect.height;
   bRect.Volume = bRect.length * bRect.height;
   if (aRect.Volume < bRect.Volume) return true;
   else return false;

}
public static Boolean operator >(Rectangle aRect, Rectangle bRect)
{
   Rectangle pRect = new Rectangle();
   aRect.Volume = aRect.length * aRect.height;
   bRect.Volume = bRect.length * bRect.height;
   if (aRect.Volume > bRect.Volume) return true;
   else return false;
}
   };

   class Program
   {

      // The main function
```

```
        static void Main(string[] args)
        {

            // variable of the type student
            // Setting the values of the Student object

            Rectangle Rect1 = new Rectangle();
            Rectangle Rect2 = new Rectangle();
            Rectangle Rect3 = new Rectangle();

            Rect1.length = 2;
            Rect1.height = 2;

            Rect2.length = 3;
            Rect2.height = 4;

            Console.WriteLine("Is Rectangle 1 less than Rectangle 2 " +
(Rect1 < Rect2));

        Console.Read();

        }
    }
}
```

With this program, the output is as follows:

Is Rectangle 1 less than Rectangle 2 True

13. Anonymous Methods

Anonymous methods provide a technique to pass a code block as a delegate parameter. Anonymous methods are methods without a name, just the body.

You don't need to specify the return type in an anonymous method, it is inferred from the return statement inside the method body. In order to use an anonymous method, we need to declare it using the "delegate" keyword.

```
delegate returndatatype methodname(methodparameters);
```

Where:

- The method is a normal method with a return data type and parameters.
- Next you can assign a variable to the delegate method and define the body of the anonymous method.
- And finally we can call the anonymous method like a normal method using the variable name.

The best way to understand this is to see this via an example shown below.

Example 80: The following program is used to showcase how to use anonymous methods.

```
using System;

namespace Demo
{
   // Declaring the delegate method
   delegate void NumberDisplay(int n);

   class Program
   {

      // The main function
```

```
        static void Main(string[] args)
        {

            // Defining the delegate method
            NumberDisplay var1 = delegate(int a)
            {
                Console.WriteLine("The integer passed to the method is "+
a);
            };

            // calling the method
            var1(10);

        Console.Read();

        }
    }
}
```

With this program, the output is as follows:

The integer passed to the method is 10

Conclusion

This has brought us to the end of this book, but it doesn’t mean your C# journey should end here. If you enjoyed this guide, be sure to keep a lookout for the next book in the series that covers more advanced topics of C#.

Lastly, this book was written not only to be a teaching guide, but also a reference manual. So remember to always keep it near, as you venture through this wonderful world of programming.

About the Author

Nathan Clark is an expert programmer with nearly 20 years of experience in the software industry.

With a master's degree from MIT, he has worked for some of the leading software companies in the United States and built up extensive knowledge of software design and development.

Nathan and his wife, Sarah, started their own development firm in 2009 to be able to take on more challenging and creative projects. Today they assist high-caliber clients from all over the world.

Nathan enjoys sharing his programming knowledge through his book series, developing innovative software solutions for their clients and watching classic sci-fi movies in his free time.