C#:
A practical guide to C# Basics and some important features
(For solving DSA Problems)

Sanfyin

Sanfy.In

Open Digital Publisher

Open Publisher
Open Digital Publisher
https://www.sanfy.in/
First edition 2022

This book is digitaly printed
Printed in the ... (Github)

This book is dedicated to Learners

Preface

This book includes C# basics and some important features availabe in the language to solve DSA problems.

Contents

1

Introduction

C# (pronounced C-Sharp) is an Object Oriented Programming language and it has many similarities to Java, C++ and VB. In fact, C# combines the power and efficiency of C++, the simple and clean OO design of Java and the language simplification of Visual Basic.

Writing Your First Hello World Program in C#

We can use any Text Editor for writting a C# program. We will use Visual Studio Code for this training.

Create a file as Program.cs, write below Code and save the file.

```
class Program
{
    public static void Main(string[] args)
    {
        System.Console.WriteLine("Hello World!");
    }
}
```

To compile this file, go to command prompt (Windows) or Terminal (Mac) and execute below command:

`csc Program.cs`

This will compile your program and create an .exe file (Program.exe) in the same directory and will report any errors that may occur.

To run your program, in the command prompt or terminal, type `Program` in Windows or `mono Program` in Mac.

This will print Hello World as a result on your console screen. Simple, isn't it?

Understanding the Hello World Application Code:

The class Keyword

All of our C# programs contain at least one class. The Main() method resides in one of these classes. Classes are a combination of data (fields) and functions (methods) that can be performed on this data in order to achieve the solution to our problem. We will see the concept of class in more detail later. Classes in C# are defined using the class keyword followed by the name of class.

The Main() Method

In the next line we defined the Main() method of our program:

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

This is the standard signature of the Main method in C#. The Main method is the entry point of our program, i.e., our C# program starts its execution from the first line of Main method and terminates with the termination of Main method.

Printing on the Console

Our next line prints Hello World on the Console screen:

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

Here we called WriteLine(), a static method of the Console class defined in the System namespace. This method takes a string (enclosed in double quotes) as its parameter and prints it on the Console window. C#, like other Object Oriented languages, uses the dot (.) operator to access the member variables (fields) and methods of a class. Also, braces () are used to identify methods in the code and string literals are enclosed in double quotation marks ("). Lastly, each statement in C# (like C, C++ and Java) ends with a semicolon (;), also called the statement terminator.

Namespaces in C#

A Namespace is simply a logical collection of related classes in C#. We bundle our related classes in some named collection calling it a namespace. As C# does not allow two classes with the same name to be used in a program, the sole purpose of using namespaces is to

prevent name conflicts. This may happen when you have a large number of classes, as is the case in the Framework Class Library (FCL). It is very much possible that our Connection Class in DataActivity conflicts with the Connection Class of InternetActivity. To avoid this, these classes are made part of their respective namespace. So the fully qualified name of these classes will be DataActivity.Connection and InternetActivity.Connection, hence resolving any ambiguity for the compiler.

We can define the namspace *Training* in our program as below:

```
1
2  namespace Training
3  {
4      class Program
5      {
6          public static void Main()
7          {
8              System.Console.WriteLine("Hello World!");
9          }
10     }
11 }
```

Now we can access Program class and Main method from anywhere in a program as below:

```
1      Training.Program.Main();
```

The using Keyword

We can write the namespaces in a using statement at the start of a program as below:

```
1      using System;
2
3      namespace Training
4      {
5          class Program
6          {
7              public static void Main()
8              {
9                  Console.WriteLine("Hello World!");
10             }
11         }
12     }
```

The *using* keyword above allows us to use the classes in the 'System' namespace. By doing this, we can now access all the classes defined in the System namespace like we are able to access the Console class in our Main method.

2

C# Language Fundamentals

2.1 Basic Data Types

There are two kinds of data types in C#.

- Value Types (implicit data types, structs and enumeration)
- Reference Types (objects, delegates)

Value types are passed to methods by passing an exact copy while Reference types are passed to methods by passing only their reference (handle). Implicit data types are defined in the language core by the language vendor, while explicit data types are types that are made by using or composing implicit data types. The commonly used implicit data types in C# are mentioned below:

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
decimal	16 bytes	Stores fractional numbers. Sufficient for storing 28 decimal digits
bool	1 byte	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

Implicit data types are represented in language using keywords, so each of the above is a keyword in C#. Keyword are the words defined by the language and can not be used as identifiers. It is worth noting that string is also an implicit data type in C#, so string is a keyword in C#. The last point about implicit data types is that they are value types and thus stored on the stack, while user defined types or referenced types are stored using the heap. A stack is a data structure that store items in a first in first out (FIFO) fashion. It is an area of memory supported by the processor and its size is determined at the compile time. A heap consists of memory available to the program at run time. Reference types are allocated using memory available from the heap dynamically (during the execution of program). The garbage collector searches for non-referenced data in heap during the execution of program and returns that space to Operating System.

2.2 Operators

Arithmetic Operators

Several common arithmetic operators are allowed in C#.

Operand	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Remainder or modulo
++	Increment by 1
–	Decrement by 1

```
using System;
namespace Training
{
    class Program
    {
        // The program shows the use of arithmetic operators
        // + - * / % ++ --
        static void Main()
        {
            // result of addition, subtraction,
            // multiplication and modulus operator
            int sum=0, difference=0, product=0, modulo=0;
            float quotient=0; // result of division
            int num1 = 10, num2 = 2; // operand variables
            sum = num1 + num2;
            difference = num1 - num2;
            product = num1 * num2;

            quotient = num1 / num2;
            // remainder of 3/2
            modulo = 3 % num2;
            Console.WriteLine("num1 = {0}, num2 = {1}", num1, num2);
            Console.WriteLine();
            Console.WriteLine ("Sum of {0} and {1} is {2}", num1,
                num2, sum);
            Console.WriteLine("Difference of {0} and {1} is {2}",
                num1, num2, difference);
            Console.WriteLine("Product of {0} and {1} is {2}", num1,
                num2, product);
            Console.WriteLine("Quotient of {0} and {1} is {2}", num1,
                num2, quotient);
            Console.WriteLine();
            Console.WriteLine("Remainder when 3 is divided by {0} is
                {1}", num2, modulo);
            num1++; // increment num1 by 1
            num2--; // decrement num2 by 1
            Console.WriteLine("num1 = {0}, num2 = {1}", num1, num2);
```

```
        }
    }
}
```

Although the program above is quite simple, I would like to discuss some concepts here. In the *Console.WriteLine()* method, we have used format-specifiers {int} to indicate the position of variables in the string.

```
    Console.WriteLine("Sum of {0} and {1} is {2}", num1, num2, sum);
```

Here, {0}, {1} and {2} will be replaced by the values of the num1, num2 and sum variables. In i, i specifies that (i+1)th variable after double quotes will replace it when printed to the Console. Hence, 0 will be replaced by the first one, {1} will be replaced by the second variable and so on...
Another point to note is that num1++ has the same meaning as:
num1 = num1 + 1;
Or:
num1 += 1;

Prefix and Postfix notation

Both the ++ and – operators can be used as prefix or postfix operators. In prefix form:

```
    num1 = 3;
    num2 = ++num1; // num1 = 4, num2 = 4
```

The compiler will first increment num1 by 1 and then will assign it to num2. While in postfix form:

```
    num2 = num1++; // num1 = 4, num2 = 3
```

The compiler will first assign num1 to num2 and then increment num1 by 1.

Assignment Operators

Assignment operators are used to assign values to variables. Common assignment operators in C# are:

Operand	Description
=	Simple assignment
+=	Additive assignment
-=	Subtractive assignment
*=	Multiplicative assignment
/=	Division assignment
%=	Modulo assignment

The equals (=) operator is used to assign a value to an object. Like we have seen

```
bool isPaid = false;
```

assigns the value 'false' to the isPaid variable of Boolean type. The left hand and right hand side of the equal or any other assignment operator must be compatible, otherwise the compiler will complain about a syntax error. Sometimes casting is used for type conversion, e.g., to convert and store a value in a variable of type double to a variable of type int, we need to apply an integer cast.

```
double doubleValue = 4.67;
// intValue will be equal to 4
int intValue = (int) doubleValue;
```

Of course, when casting there is always a danger of some loss of precision; in the case above, we only got the 4 of the original 4.67. Sometimes, the casting may result in strange values:

```
int intValue = 32800;
short shortValue = (short) intValue;
// shortValue would be equal to -32736
```

Variables of type short can only take values ranging from -32768 to 32767, so the cast above can not assign 32800 to shortValue. Hence shortValue took the last 16 bits (as a short consists of 16 bits) of the integer 32800, which gives the value -32736 (since bit 16, which represents the value 32768 in an int, now represents -32768). If you try to cast incompatible types like:

```
bool isPaid = false;
int intValue = (int) isPaid;
```

It won't get compiled and the compiler will generate a syntax error.

Relational Operators Relational operators are used for comparison purposes in conditional statements. Common relational operators in C# are:

Operand	Description
==	Equality check
!=	Un-equality check
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Relational operators always result in a Boolean statement; either true or false. For example if we have two variables

```
int num1 = 5, num2 = 6;
```

Then:

```
num1 == num2 // false
num1 != num2 // true
num1 > num2 // false
num1 < num2 // true
num1 <= num2 // true
num1 >= num2 // false
```

Only compatible data types can be compared. It is invalid to compare a bool with an int, so if you have

```
int i = 1;
bool b = true;
```

you cannot compare i and b for equality (i==b). Trying to do so will result in a syntax error.

Logical and Bitwise Operators

These operators are used for logical and bitwise calculations. Common logical and bitwise operators in C# are:

Operand	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
!	Bitwise NOT

The operators &, | and ^are rarely used in usual programming practice. The NOT operator is used to negate a Boolean or bitwise expression like:

```
bool b = false;
bool bb = !b;
// bb would be true
```

Logical Operators && and || are used to combine comparisons like

```
int i=6, j=12;
bool firstVar = i>3 && j<10;
// firstVar would be false
bool secondVar = i>3 || j<10;
// secondVar would be true
```

In the first comparison: i>3 && j<10 will result in true only if both the conditions i>3 and j<10 result in true. In the second comparison: i>3 || j<10 will result in true if any of the conditions i>3 and j<10 result in true. You can, of course, use both && and || in single statement like:

```
bool firstVar = (i>3 && j<10) || (i<7 && j>10) // firstVar would
    be true
```

In the above statement we used parenthesis to group our conditional expressions and to avoid any ambiguity. You can use & and | operators in place of && and || but for combining conditional expressions, && and || are more efficient because they use "short circuit evaluation". For example, if in the expression (i>3 && j<10), i>3 evaluates to false, the second expression j<10 won't be checked and false will be returned (when using AND, if one of the participant operands is false, the whole operation will result in false). Hence, one should be very careful when using assignment expressions with && and || operators. The & and | operators don't do short circuit evaluation and do execute all the comparisons before returning the result.

Other Operators

There are some other operators present in C#. A short description of these is given below:

Operand	Description
«	Left shift bitwise operator
»	Right shift bitwise operator
.	Member access for objects
""	Index operator used in arrays and collections
()	Cast operator
?:	Ternary operator

Operator Precedence

All operators are not treated equally. There is a concept of "operator precedence" in C#. For example:

```
int i = 2 + 3 * 6;
// i would be 20 not 30
```

3 will be multiplied by 6 first then the result will be added to 2. This is because the multiplication operator * has precedence over the addition operator +. For a complete table of operator precedence, consult MSDN or the .Net framework documentation.

2.3 Flow Control And Conditional Statements

The if...else statement

Condition checking has always been the most important construct in any language right from the time of the assembly language days. C# provides conditional statements in the form of the if...else statement. The structure of this statement is:

if(Boolean expression)
Statement or block of statements
else
Statement or block of statements

The else clause above is optional. A typical example is:

```
    if(i==5)
    {
        Console.WriteLine("Thank God, I finally became 5.");
    }
```

In the above example, the console message will be printed only if the expression i==5 evaluates to true. If you would like to take some action when the condition does not evaluate to true, then you can use else clause:

```
    if(i==5)
    {
        Console.WriteLine ("Thank God, I finally became 5.");
    }
    else
    {
        Console.WriteLine("Missed...When will I become 5?");
    }
```

Only the first message will be printed if i is equal to 5. In any other case (when i is not 5), the second message will be printed.

You can also have if after else for further conditioning:

```
    if(i==5) // line 1
    {
        Console.WriteLine("Thank God, I finally became 5.");
    }
    else if(i==6) // line 5
    {
        Console.WriteLine("Ok, 6 is close to 5.");
    }
    else // line 9
    {
        Console.WriteLine("Missed...When will I become 5 or be close
           to 5?");
    }
```

Here else if(i==6) is executed only if the first condition i==5 is false, and else at line 9 will be executed only if the second condition i==6 (line 5) executes and fails (that is, both the

first and second conditions fail). The point here is else at line 9 is related to if on line 5.

Since if...else is also a statement, you can use it under other if...else statement (nesting), like:

```
1    if(i>5) // line 1
2    {
3        if(i==6) // line 3
4        {
5            Console.WriteLine("Ok, 6 is close to 5.");
6        }
7        else // line 7
8        {
9        Console.WriteLine("Oops! I'm older than 5 but not 6!");
10       }
11       Console.WriteLine("Thank God, I finally became older than
             5.");
12   }
13   else // line 13
14   {
15       Console.WriteLine("Missed...When will I become 5 or close to
             5?");
16   }
```

The else on line 7 is clearly related to if on line 3 while else on line 13 belongs to if on line 1. Finally, do note (C/C++ programmers especially) that if statements expect only Boolean expressions and not integer values. It's an error to write:

```
1    int flag = 0;
2    if(flag = 1)
3    {
4        // do something...
5    }
```

Instead, you can either use:

```
1    int flag = 0;
2    if(flag == 1) // note ==
3    {
4        // do something...
5    }
```

or,

```
1    bool flag = false;
2    if(flag = true) // Boolean expression
3    {
4        // do something...
```

```
    }
```

The keys to avoiding confusion in the use of any complex combination of if...else are aligning the code to enhance readability. If you are using Visual Studio or some other editor that supports the language, the editor will do indentation for you. Otherwise, you have to take care of this yourself.

The switch...case statement

If you need to perform a series of specific checks, switch...case is present in C# just for this. The general structure of the switch...case statement is:

```
switch(integral or string expression)
{
    case constant-expression:
    statements
    breaking or jump statement
    // some other case blocks
    ...
    default:
    statements
    breaking or jump statement
}
```

It takes less time to use switch...case than using several if...else if statements. Let's look at it with an example:

```
using System;
class SwitchCaseExample
{
    // Demonstrates the use of switch...case statement along with
    // the use of command line argument
    static void Main(string [] userInput)
    {
        int input = int.Parse(userInput[0]);
        // convert the string input to integer.
        // Will throw a run-time exception if there is no input
           at run-time or if
        // the input is not castable to integer.
        switch(input) // what is input?
        {
            case 1: // if it is 1
            Console.WriteLine("You typed 1 (one) as the first
               command line argument");
            break; // get out of switch block
            case 2: // if it is 2
```

```
                Console.WriteLine("You typed 2 (two) as the first
                    command line argument");
                break; // get out of switch block
                case 3: // if it is 3
                Console.WriteLine("You typed 3 (three) as the first
                    command line argument");
                break; // get out of switch block
                default: // if it is not any of the above
                Console.WriteLine("You typed a number other than 1, 2
                    and 3");
                break; // get out of switch block
            }
        }
    }
```

The program must be supplied with an integer command line argument. First, compile the program (at the command line or in Visual Studio.Net). Suppose we made an exe with name "SwitchCaseExample.exe", we would run it at the command line like this:
Let's get to internal working. First, we converted the first command line argument (user-Input[0]) into an int variable input. For conversion, we used the static Parse() method of the int data type. This method takes a string and returns the equivalent integer or raises an exception if it can't. Next we checked the value of input variable using a switch statement:

```
    switch(input)
    {
    ...
    }
```

Later, on the basis of the value of input, we took specific actions under respective case statements. Once our case specific statements end, we mark it with the break statement before the start of another case (or the default) block.

```
    case 3: // if it is 3
    Console.WriteLine("You typed 3 (three) as first command line
        argument");
    break; // get out of switch block
```

If all the specific checks fail (input is none of 1,2 and 3), the statements under default executes.

```
    default:
    // if it is not any of the above
    Console.WriteLine ("You typed a number other than 1, 2 and 3");
    break; // get out of switch block
```

There are some important points to remember when using the switch...case statement in C#:

- You can use either integers (enumeration) or strings in a switch statement
- The expression following case must be constant. It is illegal to use a variable after case: case i: // incorrect, syntax error
- You can use multiple statements under single case and default statements:
- A colon : is used after the case statement and not a semicolon ; case "India": continent = "Asia"; Console.WriteLine("India is an Asian Country"); break; default: continent = "Un-recognized"; Console.WriteLine ("Un-recognized country discovered"); break;
- The end of the case and default statements is marked with break (or goto) statement. We don't use brackets to mark the block in switch...case as we usually do in C#
- C# does not allow fall-through. So, you can't leave case or default without break statement (as you can in Java or C/C++). The compiler will detect and complain about the use of fall-through in the switch...case statement.
- The break statement transfers the execution control out of the current block.
- Statements under default will be executed if and only if all the case checks fail.
- It is not necessary to place default at the end of switch...case statement. You can even place the default block before the first case or in between cases; default will work the same regardless of its position. However, making default the last block is conventional and highly recommended. Of course, you can't have more than one default block in a single switch...case.

Loops In C#

Loops are used for iteration purposes, i.e., doing a task multiple times (usually until a termination condition is met)

The for Loop

The most common type of loop in C# is the for loop. The basic structure of a for loop is exactly the same as in Java and C/C++ and is:

```
for(assignment; condition; increment/decrement)
    statements or block of statements
    enclosed in {} brackets
```

Let's see a for loop that will write the integers from 1 to 10 to the console:

```
for(int i=1; i<=10; i++)
{
    Console.WriteLine("In the loop, the value of i is {0}.", i);
}
```

At the start, the integer variable i is initialized with the value of 1. The statements in the for loop are executed while the condition (i<=10) remains true. The value of i is incremented (i++) by 1 each time the loop starts.

Some important points about the for loop

All three statements in for(), assignment, condition and increment/decrement are optional. You can use any combination of these and even decide not to use any one at all. This would make what is called and 'Indefinite or infinite loop' (a loop that will never end until or unless the break instruction is encountered inside the body of the loop). But, you still have to supply the appropriate semi colons:

```
for(; ;)
for( ; i<10; i++)
for(int i=3; ; i--)
for( ; i>5; )
```

If you don't use the brackets, the statement immediate following for() will be treated as the iteration statement. The example below is identical to the one given above:

```
for(int i=1; i<=10; i++)
    Console.WriteLine("In the loop, value of i is {0}.", i);
```

I will again recommend that you always use brackets and proper indentation. If you declare a variable in for()'s assignment, its life (scope) will only last inside the loop and it will die after the loop body terminates (unlike some implementations of C++). Hence if you write:

```
for(int i=1; i<=10; i++)
{
    Console.WriteLine("In the loop, value of i is {0}.", i);
}
i++; // line 1
```

The compiler will complain at line 1 that "i is an undeclared identifier". You can use break and continue in for loops or any other loop to change the normal execution path. break terminates the loop and transfers the execution to a point just outside the for loop:

```
for(int i=1; i<=10; i++)
{
    if(i>5)
    {
        break;
    }
    Console.WriteLine("In the loop, value of i is {0}.", i);
}
```

The loop will terminate once the value of i gets greater than 5. If some statements are present after break, break must be enclosed under some condition, otherwise the lines following break will become unreachable and the compiler will generate a warning (in Java, it's a syntax error).

```
    for(int i=3; i<10; i++)
    {
        break; // warning, Console.WriteLine (i); is unreachable code
        Console.WriteLine(i);
    }
```

continue ignores the remaining part of the current iteration and starts the next iteration.

```
    for(int i=1; i<=10; i++)
    {
        if(i==5)
        {
            continue;
        }
        Console.WriteLine("In the loop, value of i is {0}.", i);
    }
```

Console.WriteLine will be executed for each iteration except when the value of i becomes 5. The sample output of the above code is:

```
    In the loop, value of i is 1.
    In the loop, value of i is 2.
    In the loop, value of i is 3.
    In the loop, value of i is 4.
    In the loop, value of i is 6.
    In the loop, value of i is 7.
    In the loop, value of i is 8.
    In the loop, value of i is 9.
    In the loop, value of i is 10.
```

The do...while Loop

The general structure of a do...while loop is

```
    do
        Statement or block of statements
    while(boolean expression);
```

The statements under do will execute the first time and then the condition is checked. The loop will continue while the condition remains true.

The program for printing the integers 1 to 10 to the console using the do...while loop is:

```
int i=1;
do
{
    Console.WriteLine("In the loop, value of i is {0}.", i);
    i++;
} while(i<=10);
```

Some important points here are:

- The statements in a do...while() loop always execute at least once.
- There is a semicolon ; after the while statement.

while Loop

The while loop is similar to the do...while loop, except that it checks the condition before entering the first iteration (execution of code inside the body of the loop). The general form of a while loop is:

```
while(Boolean expression)
    statements or block of statements
```

Our program to print the integers 1 to 10 to the console using while will be:

```
int i=1;
while(i<=10)
{
    Console.WriteLine("In the loop, value of i is {0}.", i);
    i++;
}
```

2.4 Arrays in C#

Array Declaration

An Array is a collection of values of a similar data type. Technically, C# arrays are a reference type. Each array in C# is an object and is inherited from the System.Array class. Arrays are declared as:

```
<data type> [] <identifier> = new <data type>[<size of array>];
```

Let's define an array of type int to hold 10 integers.

```
int [] integers = new int[10];
```

The size of an array is fixed and must be defined before using it. However, you can use variables to define the size of the array:

```
int size = 10;
int [] integers = new int[size];
```

You can optionally do declaration and initialization in separate steps:

```
int [] integers;
integers = new int[10];
```

It is also possible to define arrays using the values it will hold by enclosing values in curly brackets and separating individual values with a comma:

```
int [] integers = {1, 2, 3, 4, 5};
```

This will create an array of size 5, whose successive values will be 1, 2, 3, 4 and 5.

Accessing the values stored in an array

To access the values in an Array, we use the indexing operator [int index]. We do this by passing an int to indicate which particular index value we wish to access. It's important to note that index values in C# start from 0. So if an array contains 5 elements, the first element would be at index 0, the second at index 1 and the last (fifth) at index 4.
The following lines demonstrate how to access the 3rd element of an array:

```
int [] intArray = {5, 10, 15, 20};
int j = intArray[2];
```

Let's make a program that uses an integral array.

```
// demonstrates the use of arrays in C#
static void Main()
{
    // declaring and initializing an array of type integer
    int [] integers = {3, 7, 2, 14, 65};
    // iterating through the array and printing each element
    for(int i=0; i<5; i++)
    {
        Console.WriteLine(integers[i]);
    }
}
```

Here we used the for loop to iterate through the array and the Console.WriteLine() method to print each individual element of the array. Note how the indexing operator [] is used.
The above program is quite simple and efficient, but we had to hard-code the size of the array in the for loop. As we mentioned earlier, arrays in C# are reference type and are a

sub-class of the System.Array Class. This class has lot of useful properties and methods that can be applied to any instance of an array that we define. Properties are very much like the combination of getter and setter methods in common Object Oriented languages. Properties are context sensitive, which means that the compiler can un-ambiguously identify whether it should call the getter or setter in any given context. We will discuss properties in detail in the coming lessons. System.Array has a very useful read-only property named Length that can be used to find the length, or size, of an array programmatically.

Using the Length property, the for loop in the above program can be written as:

```
    for(int i=0; i<integers.Length; i++)
    {
        Console.WriteLine(integers[i]);
    }
```

This version of looping is much more flexible and can be applied to an array of any size and of any data-type.

Now we can understand the usual description of Main(). Main is usually declared as:

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

The command line arguments that we pass when executing our program are available in our programs through an array of type string identified by the args string array.

foreach Loop

There is another type of loop that is very simple and useful to iterate through arrays and collections. This is the foreach loop. The basic structure of a foreach loop is:

```
    foreach(<type of elements in collection> <identifier> in <array
       or collection>)
        <statements or block of statements>
```

Let's now make our previous program to iterate through the array with a foreach loop:

```
    // demonstrates the use of arrays in C#
    static void Main()
    {
        // declaring and initializing an array of type integer
        int [] integers = {3, 7, 2, 14, 65};
        // iterating through the array and printing each element
        foreach(int i in integers)
        {
            Console.WriteLine(i);
        }
```

```
11      }
```

Simple and more readable, isn't it? In the statement:

```
1       foreach(int i in integers)
```

We specified the type of elements in the collection (int in our case). We declared the variable (i) to be used to hold the individual values of the array 'integers' in each iteration.

Important points to note here:

- The variable used to hold the individual elements of array in each iteration (i in the above example) is read only. You can't change the elements in the array through it. This means that foreach will only allow you to iterate through the array or collection and not to change the contents of it. If you wish to perform some work on the array to change the individual elements, you should use a for loop.
- foreach can be used to iterate through arrays or collections. By a collection, we mean any class, struct or interface that implements the IEnumerable interface. (Just go through this point and re-read it once we complete the lesson describing classes and interfaces)
- The string class is also a collection of characters (implements IEnumerable interface and returns char value in Current property). The following code example demonstrates this and prints all the characters in the string.

```
1       static void Main()
2       {
3           string name = "Sanfy In";
4           foreach(char ch in name)
5           {
6               Console.WriteLine(ch);
7           }
8       }
```

This will print each character of the name in a separate line.

2.5 Classes and Objects

Concept of a Class

A class is simply an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be

performed on data (methods) and accessors to data (properties). For example, there is a class String in the System namespace of .Net Library. This class contains an array of characters (data) and provide different operations (methods) that can be applied to its data like ToLowerCase(), Trim(), Substring(), etc. It also has some properties like Length (used to find the length of the string).

A class in C# is declared using the keyword class and its members are enclosed in parenthesis

```
class MyClass
{
    // fields, operations and properties go here
}
```

where MyClass is the name of class or new data type that we are defining here.

Objects

As mentioned above, a class is an abstract model. An object is the concrete realization or instance built on the model specified by the class. An object is created in the memory using the keyword 'new' and is referenced by an identifier called a "reference".

```
MyClass myObjectReference = new MyClass();
```

In the line above, we made an object of type MyClass which is referenced by an identifier myObjectReference. The difference between classes and implicit data types is that objects are reference types (passed by reference) while implicit data types are value type (passed by making a copy). Also, objects are created at the heap while implicit data types are stored on stack.

Fields

Fields are the data contained in the class. Fields may be implicit data types, objects of some other class, enumerations, structs or delegates. In the example below, we define a class named Student containing a student's name, age, marks in maths, marks in English, marks in science, total marks, obtained marks and a percentage.

```
class Student
{
    // fields contained in Student class
    string name;
    int age;
    int marksInMaths;
    int marksInEnglish;
```

```
        int marksInScience;
        int totalMarks = 300; // initialization
        int obtainedMarks;
        double percentage;
    }
```

You can also initialize the fields with the initial values as we did in totalMarks in the example above. If you don't initialize the members of the class, they will be initialized with their default values.

Default values for different data types are shown below:

Data Type	Default Value
int	0
long	0
float	0.0
double	0.0
bool	False
char	(null character)
string	"" (empty string)
Objects	null

Methods

Methods are the operations performed on the data. A method may take some input values through its parameters and may return a value of a particular data type. The signature of the method takes the form

```
    <return type> <name of method>(<data type> <identifier>, <data
        type> <identifier>,...)
    {
        // body of the method
    }
```

For example,

```
    int FindSum(int num1, int num2)
    {
        int sum = num1 + num2;
        return sum;
    }
```

Here, we defined a method named FindSum which takes two parameters of int type (num1 and num2) and returns a value of type int using the keyword return. If a method does not return anything, its return type would be void. A method can also optionally take no parameter (a parameterless method)

```
    void ShowCurrentTime()
    {
        Console.WriteLine("The current time is: " + DateTime.Now);
    }
```

The above method takes no parameter and returns nothing. It only prints the Current Date and Time on the console using the DateTime Class in the System namespace.

Instantiating the class

In C# a class is instantiated (making its objects) using the new keyword.

```
    Student theStudent = new Student();
```

You can also declare the reference and assign an object to it in different steps. The following two lines are equivalent to the above line

```
    Student theStudent;
    theStudent = new Student();
```

Note that it is very similar to using implicit data types except for the object is created with the new operator while implicit data types are created using literals

```
    int i;
    i = 4;
```

Another important thing to understand is the difference between reference and object. The line

```
    Student theStudent;
```

only declares the reference theStudent of type Student which at this point does not contain any object (and points to the default null value) so if you try to access the members of class (Student) through it, it will throw a compile time error 'Use of unassigned variable theStudent'. When we write

```
    theStudent = new Student();
```

then a new object of type Student is created at the heap and its reference (or handle) is given to theStudent. Only now is it legal to access the members of the class through it.

Accessing the members of a class

The members of a class (fields, methods and properties) are accessed using dot '.' operator against the reference of the object like this:

```
Student theStudent = new Student();
theStudent.marksOfMaths = 93;
theStudent.CalculateTotal();
Console.WriteLine(theStudent.obtainedMarks);
```

Let's now make our Student class with some related fields, methods and then instantiate it in the Main() method.

```
using System;
namespace CSharpSchool
{
    // Defining a class to store and manipulate students information
    class Student
    {
        // fields
        string name;
        int age;
        int marksOfMaths;
        int marksOfEnglish;
        int marksOfScience;
        int totalMarks = 300;
        int obtainedMarks;
        double percentage;
        // methods
        void CalculateTotalMarks()
        {
            obtainedMarks = marksOfMaths + marksOfEnglish + marksOfScience;
        }
        void CalculatePercentage()
        {
            percentage = (double) obtainedMarks / totalMarks * 100;
        }
        double GetPercentage()
        {
            return percentage;
        }
        // Main method or entry point of program
        static void Main()
        {
            // creating new instance of Student
            Student st1 = new Student();
            // setting the values of fields
            st1.name = "Einstein";
            st1.age = 20;
```

```
                st1.marksOfEnglish = 80;
                st1.marksOfMaths = 99;
                Programmers Heaven: C# School
                58
                st1.marksOfScience = 96;
                // calling functions
                st1.CalculateTotalMarks();
                st1.CalculatePercentage();
                double st1Percentage = st1.GetPercentage();
                // calling and retrieving value
                // returned by the function
                Student st2 = new Student();
                st2.name = "Newton";
                st2.age = 23;
                st2.marksOfEnglish = 77;
                st2.marksOfMaths = 100;
                st2.marksOfScience = 99;
                st2.CalculateTotalMarks();
                st2.CalculatePercentage();
                double st2Percentage = st2.GetPercentage();
                Console.WriteLine("{0} of {1} years age got {2}%
                    marks", st1.name, st1.age, st1.percentage);
                Console.WriteLine("{0} of {1} years age got {2}%
                    marks", st2.name, st2.age, st2.percentage);
            }
        }
    }
```

Here, we started by creating an object of the Student class (st1), we then assigned name, age and marks of the student. Later, we called methods to calculate totalMarks and percentage, then we retrieved and stored the percentage in a variable and finally printed these on a console window.

We repeated the same steps again to create another object of type Student, set and printed its attributes. Hence in this way, you can create as many object of the Student class as you want. When you compile and run this program it will display:

```
    Einstein of 20 years age got 91.6666666666667% marks
    Newton of 23 years age got 92% marks
```

Access Modifiers or Accessibility Levels

In our Student class, everyone has access to each of the fields and methods. So if one wants, he/she can change the totalMarks from 300 to say 200, resulting in the percentages getting beyond 100%, which in most cases we like to restrict. C# provides access modifiers

or accessibility levels just for this purpose, i.e., restricting access to a particular member. There are 5 access modifiers that can be applied to any member of the class. We are listing these along with short description in the order of decreasing restriction

Access Modifier	Description
private	private members can only be accessed within the class that contains them
protected internal	This type of member can be accessed from the current project or from the types inherited from their containing type
internal	Can only be accessed from the current project
protected	Can be accessed from a containing class and types inherited from the containing class
public	public members are not restricted to anyone. Anyone who can see them can also access them.

In Object Oriented Programming (OOP) it is always advised and recommended to mark all your fields as private and allow the user of your class to access only certain methods by making them public. For example, we may change our student class by marking all the fields private and the three methods in the class public.

```
class Student
{
    // fields
    private string name;
    private int age;
    private int marksOfMaths;
    private int marksOfEnglish;
    private int marksOfScience;
    private int totalMarks = 300;
    private int obtainedMarks;
    private double percentage;
    // methods
    public void CalculateTotalMarks()
    {
        obtainedMarks = marksOfMaths + marksOfEnglish +
            marksOfScience;
    }
    public void CalculatePercentage()
    {
        percentage = (double) obtainedMarks / totalMarks * 100;
    }
    public double GetPercentage()
```

```
22          {
23              return percentage;
24          }
25      }
```

If you don't mark any member of class with an access modifier, it will be treated as a private member; this means the default access modifier for the members of a class is private.

You can also apply access modifiers to other types in C# such as the class, interface, struct, enum, delegate and event. For top-level types (types not bound by any other type except namespace) like class, interface, struct and enum you can only use public and internal access modifiers with the same meaning as described above. In fact other access modifiers don't make sense to these types. Finally you can not apply access modifiers to namespaces.

Properties

You must be wondering if we declare all the fields in our class as private, how can we assign values to them through their reference as we did in the Student class before? The answer is through Properties. C# is the first language to provide the support of defining properties in the language core.

In traditional languages like Java and C++, for accessing the private fields of a class, public methods called getters (to retrieve the value) and setters (to assign the value) were defined like if we have a private field name private string name; then, the getters and setters would be like

```
1       // getter to name field
2       public string GetName()
3       {
4           return name;
5       }
6       // setter to name field
7       public void SetName(string theName)
8       {
9           name = theName;
10      }
```

Using these we could restrict the access to a particular member. For example we can opt to only define the getter for the totalMarks field to make it read only.

```
1       private int totalMarks;
2       public int GetTotalMarks()
3       {
4           return totalMarks;
```

```
    }
```

Hence outside the class, one can only read the value of totalMarks and can not modify it. You can also decide to check some condition before assigning a value to your field

```
    private int marksOfMaths;
    public void SetMarksOfMaths(int marks)
    {
        if(marks >= 0 && marks <=100)
        {
            marksOfMaths = marks;
        }
        else
        {
            marksOfMaths = 0;
            // or throw some exception informing user marks out of
               range
        }
    }
```

This procedure gives you a lot of control over how fields of your classes should be accessed and dealt in a program. But, the problem is this you need to define two methods and have to prefix the name of your fields with Get or Set. C# provides the built in support for these getters and setters in the form of properties. Properties are context sensitive constructs used to read, write or compute private fields of class and to achieve control over how the fields can be accessed.

Using Properties

The general Syntax for Properties is

```
    <access modifier> <data type> <name of property>
    {
        get
        {
            // some optional statements
            return <some private field>;
        }
        set
        {
            // some optional statements;
            <some private field> = value;
        }
    }
```

Didn't understand it? No problem. Let's clarify it with an example: we have a private field name

```
    private string name;
```

We decide to define a property for this providing both getters and setters. We will simply write

```
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
```

We defined a property called 'Name' and provided both a getter and a setter in the form of get and set blocks. Note that we called our property 'Name' which is accessing the private field 'name'. It is becoming convention to name the property the same as the corresponding field but with first letter in uppercase (for name->Name, for percentage->Percentage). As properties are accessors to certain fields, they are mostly marked as public while the corresponding field is (and should be) mostly private. Finally note in the set block, we wrote

```
    name = value;
```

Here, value is a keyword and contains the value passed when a property is called. In our program we will use our property as

```
    Student theStudent = new Student();
    theStudent.Name = "Sanfy";
    string myName = theString.Name;
    theStudent.name = "Someone not Sanfy"; // error
```

While defining properties, we said properties are context sensitive. When we write

```
    theStudent.Name = "Sanfy";
```

The compiler sees that the property Name is on the left hand side of assignment operator, so it will call the set block of the properties passing "Sanfy" as a value (which is a keyword). In the next line when we write

```
    string myName = theString.Name;
```

the compiler now sees that the property Name is on the right hand side of the assignment operator, hence it will call the get block of property Name which will return the contents of the private field name ("Sanfy" in this case, as we assigned in line 2) which will be stored in the local string variable name. Hence, when compiler finds the use of a property, it checks in which context it is called and takes appropriate action with respect to the context.
The last line

```
1     theStudent.name = "Someone not Faraz"; // error
```

will generate a compile time error (if called outside the Student class) as the name field is declared private in the declaration of class.

You can give the definition of either of get or set block. If you miss one of these, and user tries to call it, he/she will get compile time error. For example the Length property in String class is read only; that is, the implementers have only given the definition of get block. You can write statements in the get , set blocks as you do in methods.

```
1     private int marksOfMaths;
2     public int MarksOfMaths
3     {
4         set
5         {
6             if(value >= 0 && value<=100)
7             {
8             marksOfMaths = value;
9             }
10            else
11            {
12            marksOfMaths = 0;
13            // or throw some exception informing user marks out of
                  range
14            }
15        }
16    }
```

Precautions when using properties

- Properties don't have argument lists; set, get and value are keywords in C#
- The data type of value is the same as the type of property you declared when declaring the property
- ALWAYS use proper curly brackets and proper indentation while using properties.
- DON'T try to write the set or get block in a single line

- UNLESS your property only assigns and retrieve values from the private fields like
  get return name;
  set name = value;

Each object has a reference this which points to itself. Suppose in some method call, our object needs to pass itself, what would we do? Suppose in our class Student, we have a method Store() that stores the information of Student on the disk. In this method, we called another method Save() of FileSystem class which takes the object to store as its parameter.

```
class Student
{
    string name = "Some Student";
    int age;
    public void Store()
    {
        FileSystem fs = new FileSystem();
        fs.save(this);
    }
}
```

We passed this as a parameter to the method Save() which points to the object itself.

```
class Test
{
    public static void Main()
    {
        Student theStudent = new Student();
        theStudent.Store();
    }
}
```

Here, when Store() is called, the reference theStudent will be passed as a parameter to the Save() method in Store(). Conventionally, the parameters to constructors and other methods are named the same as the name of the fields they refer to and are distinguished only by using this reference.

```
class Student
{
    private string name;
    private int age;
    public Student(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

Here in the constructor when we use name or age, we actually get the variables passed in the method which overshadow the instance members (fields) with same name. Hence, to use our fields, we had to use this to distinguish our instance members (fields) with the members passed through the parameters.
This is an extremely useful, widely and commonly used construct. I recommend you practice with "this" for some time until you feel comfortable with it.

3

DSA Specifics

3.1 Arrays

Arrays Revisited

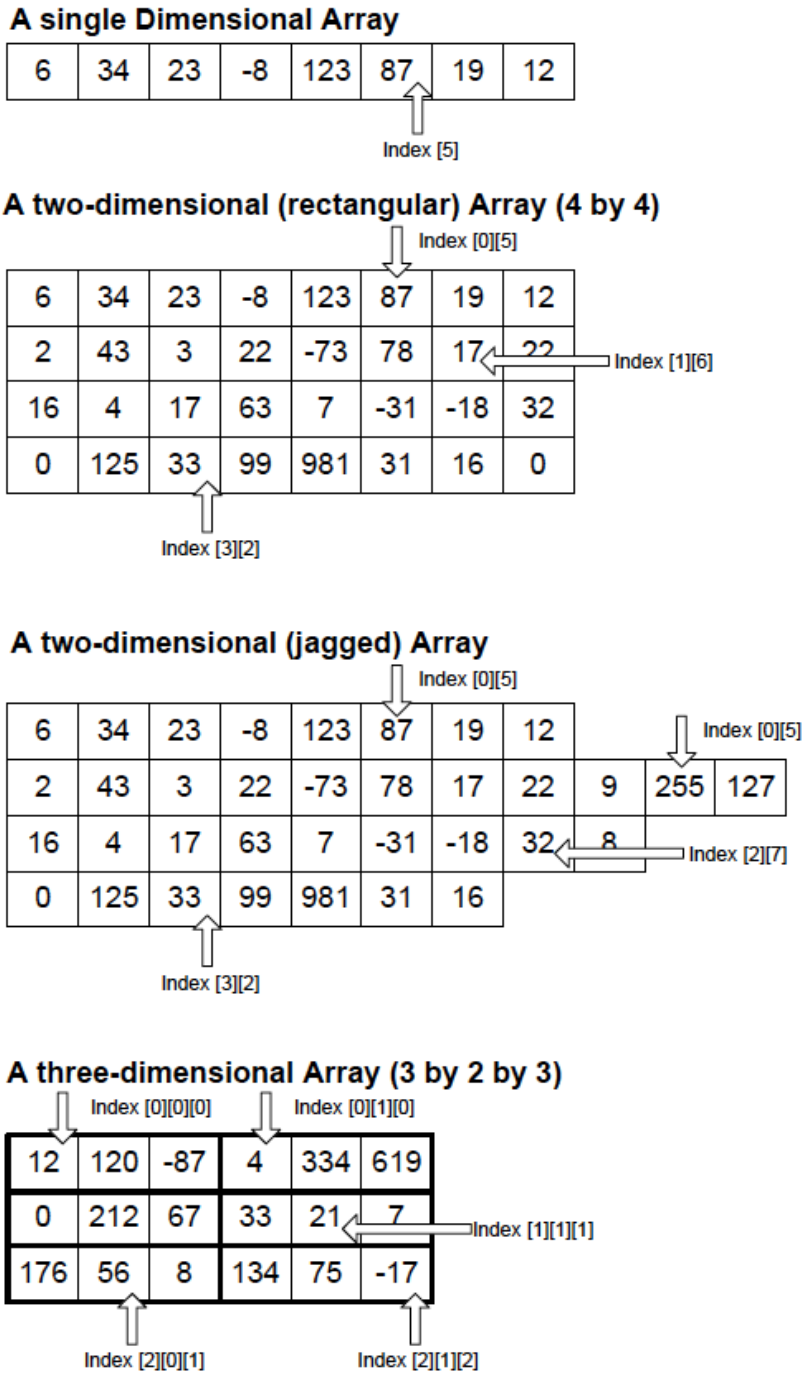
As we have seen earlier, an array is a sequential collection of elements of a similar data type. In C#, an array is an object and thus a reference type, and therefore they are stored on the heap. We have only covered single dimension arrays in the previous lessons, now we will explore multidimensional arrays.

Multidimensional Arrays

A multidimensional array is an 'array of arrays'. A multidimensional array is the one in which each element of the array is an array itself. It is similar to tables in a database where each primary element (row) is a collection of secondary elements (columns). If the secondary elements do not contain a collection of other elements, it is called a 2-dimensional array (the most common type of multidimensional array), otherwise it is called an n-dimensional array where n is the depth of the chain of arrays. There are two types of multidimensional arrays in C#:

- Rectangular array (one in which each row contains an equal number of columns)
- Jagged array (one in which each row does not necessarily contain an equal number of columns)

The images below show what the different kinds of arrays look like. The figure also shows the indexes of different elements of the arrays. Remember, the first element of an array is always zero (0).



Instantiating and accessing the elements of multidimensional arrays

Recall that we instantiate our single dimensional array like this:

```
1    int [] intArray = new int[5];
```

The above line would instantiate (create) a one dimensional array (intArray) of type int, and with 5 elements. We can access the elements of the array like this:

```
intArray[0] = 45; // set the first element to 45
intArray[2] = 21; // set the third element to 21
intArray[4] = 9; // set the fifth and last element to 9
```

Instantiating a multidimensional array is almost identical to the above procedure, as long as you keep the most basic definition of the multidimensional array in mind which says 'a multidimensional array is an array of arrays'. Suppose we wish to create a two dimensional rectangular array with 2 rows and 3 columns. We can instantiate the array as follows:

```
int [,] myTable = new int[2,3];
```

All the elements of the array are auto-initialized to their default values; hence all the elements of the myTable array would be initialized with zeroes. We can iterate through this array using either a foreach loop or a for loop.

```
foreach(int intVal in myTable)
{
    Console.WriteLine(intVal);
}
```

When it is compiled and executed, it will print six (2 x 3) zeros.

```
0
0
0
0
0
0
```

Now, let us change the values of the individual elements of the array. To change the value of the first element of the first row to 32, we can write the following code:

```
myTable[0,0] = 32;
```

In the same way we can change the values of other elements of the array:

```
myTable[0,1] = 2;
myTable[0,2] = 12;
myTable[1,0] = 18;
myTable[1,1] = 74;
myTable[1,2] = -13;
```

Now, we can use a couple of nested for loops to iterate over the array:

```
for(int row=0; row<myTable.GetLength(0); row++)
```

```
    {
        for(int col=0; col<myTable.GetLength(1); col++)
        {
            Console.WriteLine("Element at {0},{1} is {2}", row, col,
                myTable[row,col]);
        }
    }
```

Here, we have used two for loops to iterate through each of the two dimensions of the array. We have used the GetLength() method of the System.Array class (the underlying class for arrays in .Net) to find the length of a particular dimension of an array. Note that the Length property will give total number of elements in this two dimensional array, i.e., 6. The output of the above program will be:

```
Element at 0,0 is 3
Element at 0,1 is 2
Element at 0,2 is 12
Element at 1,0 is 18
Element at 1,1 is 74
Element at 1,2 is -13
```

Instantiating and accessing Jagged Arrays

A jagged array is one in which the length of each row is not the same. For example we may wish to create a table with 3 rows where the length of the first row is 3, the second row is 5 and the third row is 2. We can instantiate this jagged array like this:

```
int [][] myTable = new int[3][];
myTable[0] = new int[3];
myTable[1] = new int[5];
myTable[2] = new int[2];
```

Then we can fill the array like this:

```
myTable[0][0] = 3;
myTable[0][1] = -2;
myTable[0][2] = 16;
myTable[1][0] = 1;
myTable[1][1] = 9;
myTable[1][2] = 5;
myTable[1][3] = 6;
myTable[1][4] = 98;
myTable[2][0] = 19;
myTable[2][1] = 6;
```

Now, we will show how to use the foreach loop to access the elements of the array:

```
foreach(int []row in myTable)
```

```
    {
        foreach(int col in row)
        {
            Console.WriteLine(col);
        }
        Console.WriteLine();
    }
```

The code above is very simple and easily understandable. We picked up each row (which is an int array) and then iterated through the row while printing each of its columns. The output of the above code will be:

```
3
-2
16
1
9
5
6
98
19
6
```

In the same way, we can use a three-dimensional array:

```
int [,,] myTable = new int[3,2,4];
myTable[0,0,0] = 3;
myTable[1,1,1] = 6;
```

Or in jagged array fashion:

```
int [][][] myTable = new int[2][][];
myTable[0] = new int[2][];
myTable[0][0] = new int[3];
myTable[0][1] = new int[4];
myTable[1] = new int[3][];
myTable[1][0] = new int[2];
myTable[1][1] = new int[4];
myTable[1][2] = new int[3];
myTable[0][0][0] = 34;
myTable[0][1][1] = 43;
myTable[1][2][2] = 76;
```

Here, we have created a three dimensional jagged array. It is an array of two 2-dimensional arrays. The first of the 2-dimensional arrays contains 2 rows. The length of the first row is 3, while the length of second row is 4. In a similar fashion, the second two dimensional array is also initialized. In the end, we have accessed some of the elements of the array and assigned them different values. Although higher dimensional jagged arrays are quite

difficult to perceive; they may be very useful in certain complex problems. Again, the key to avoid confusion in multidimensional arrays is to perceive them as an 'array of arrays'.

Some other important points about multidimensional arrays

In the examples we just saw, we have used multidimensional arrays of only the integer type. But, you can declare arrays of any data type. For example, you may define an array of strings or even an array of objects of your own class.

```
string []names = new string[4];
```

You can initialize the elements of an array on the fly in the following ways

```
string []names = new string[4]{"Faraz", "Gates", "Hejlsberg",
   "Gosling"};
```

or

```
string []names = new string[]{"Faraz", "Gates", "Hejlsberg",
   "Gosling"};
```

or

```
string []names = {"Faraz", "Gates", "Hejlsberg", "Gosling"};
```

You can also separate the declaration and initialization of an array reference and the array:

```
string []names;
names = new string[4]{"Faraz", "Gates", "Hejlsberg", "Gosling"};
```

You can also initialize two-dimensional arrays on the fly (along with the declaration)

```
string [][]nameLists = {new string[]{"Sanfy", "In"},
                        new string[]{"Hejlsberg", "Gosling",
                           "Bjarne"}};
```

Some of the more important properties & methods that can be applied on arrays are:

- Length gives the number of elements in all dimensions of an array
- GetLength(int) gives the number of elements in a particular dimension of an array
- GetUpperBound() gives the upper bound of the specified dimension of an array
- GetLowerBound() gives the lower bound of the specified dimension of an array

The foreach Loop

We have been using the foreach loop for quite a long time now. Let us see how it works and how can we enable our class to be iterated over by the foreach loop. For this, we need to implement the IEnumerable interface which contains only a single method, GetEnumerator(), that returns an object of type IEnumerator. The IEnumerator interface contains one public property (Current) and two public methods MoveNext() and Reset(). The Current property is declared in the IEnumerator interface as:

```
object Current { get; }
```

It returns the current element of the collection which is of object data type. The MoveNext() method is declared as:

```
bool MoveNext();
```

It advances the current selection to the next element of the collection and returns true if the advancement is successful, and false if the collection has ended. When MoveNext() is called for the first time, it sets the selection to the first element in the collection which means that the Current property is not valid until MoveNext() has been executed for the first time.

Finally, the Reset() method is declared as

```
void Reset();
```

This method resets the enumerator and sets it to the initial state. After reset, MoveNext() will again advance the selection to the first element in the collection.
Now we will show how to make a class that can be iterated over using the foreach loop, by implementing the IEnumerable and IEnumerator interfaces.

```
using System;
using System.Collections;
namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            MyList list = new MyList();
            foreach(string name in list)
            {
                Console.WriteLine(name);
            }
        }
    }
    class MyList : IEnumerable
    {
```

```
18              static string []names = {"Sanfy", "In", "Hejlsberg",
                    "Gosling", "Bjarne"};
19              public IEnumerator GetEnumerator()
20              {
21                  return new MyEnumerator();
22              }
23
24              private class MyEnumerator : IEnumerator
25              {
26                  int index = -1;
27                  public object Current
28                  {
29                      get { return names[index]; }
30                  }
31                  public bool MoveNext()
32                  {
33                      if(++index >= names.Length)
34                          return false;
35                      else
36                          return true;
37                  }
38                  public void Reset()
39                  {
40                      index = -1;
41                  }
42              }
43          }
44      }
```

Here we have declared a class called MyList which contains a private nested class named MyEnumerator. The class MyEnumerator implements IEnumerator by providing the implementation of its public property and methods. The class MyList implements the IEnumerable interface and returns an object of type MyEnumerator in its GetEnumerator() method. The class MyList contains a static array of strings called names. The MyEnumerator class iterates through this array. It uses the integer variable index to keep track of the current element of the collection. The variable index is initialized with -1. Each time the MoveNext() method is called, it increments it by 1 and returns true or false, depending on whether collection's final element has been reached. The Current property returns the indexth element of the collection while the Reset() method resets the variable index to -1 again. In the Test class we have instantiated the MyList class and iterated through it using a foreach loop, because we have implemented IEnumerable interface on the MyList class. The output of the above program is:

```
1       Sanfy
2       In
```

```
    Hejlsberg
    Gosling
    Bjarne
```

3.2 Collections

Although we can make collections of related objects using arrays, there are some limitations when using arrays for collections. The size of an array is always fixed and must be defined at the time of instantiation of an array. Secondly, an array can only contain objects of the same data type, which we need to define at the time of its instantiation. Also, an array does not impose any particular mechanism for inserting and retrieving the elements of a collection. For this purpose, the creators of C# and the .Net Framework Class Library (FCL) have provided a number of classes to serve as a collection of different types. These classes are present in the System.Collections namespace.

Some of the most common classes from this namespace are:

Class	Description
ArrayList	Provides a collection similar to an array, but that grows dynamically as the number of elements change.
Stack	A collection that works on the Last In First Out (LIFO) principle, i.e., the last item inserted is the first item removed from the collection.
Queue	A collection that works on the First In First Out (FIFO) principle, i.e., the first item inserted is the first item removed from the collection.
HashTable	Provides a collection of key-value pairs that are organized based on the hash code of the key.
SortedList	Provides a collection of key-value pairs where the items are sorted according to the key. The items are accessible by both the keys and the index.

All of the above classes implement the ICollection interface, which contains three properties and one method:

- The Count property returns the number of elements in the collection (similar to the Length property of an Array)
- The IsSynchronized property returns a boolean value depending on whether access to the collection is thread-safe or not

- The SyncRoot property returns an object that can be used to synchronize access to the collection.
- The CopyTo(Array array, int index) method copies the elements of the collection to the array, starting from the specified index.
- All the collection classes also implement the IEnumerable interface, so they can be iterated over using the foreach loop.

The ArrayList class

The System.Collections.ArrayList class is similar to arrays, but can store elements of any data type. We don't need to specify the size of the collection when using an ArrayList (as we used to do in the case of simple arrays). The size of the ArrayList grows dynamically as the number of elements it contains changes. An ArrayList uses an array internally and initializes its size with a default value called Capacity. As the number of elements increase or decrease, ArrayList adjusts the capacity of the array accordingly by making a new array and copying the old values into it. The Size of the ArrayList is the total number of elements that are actually present in it while the Capacity is the number of elements the ArrayList can hold without instantiating a new array. An ArrayList can be constructed like this:

```
1    ArrayList list = new ArrayList();
```

We can also specify the initial Capacity of the ArrayList by passing an integer value to the constructor:

```
1    ArrayList list = new ArrayList(20);
```

We can also create an ArrayList with some other collection by passing the collection in the constructor:

```
1    ArrayList list = new ArrayList(someCollection);
```

We add elements to the ArrayList by using its Add() method. The Add() method takes an object of type object as its parameter.

```
1    list.Add(45);
2    list.Add(87);
3    list.Add(12);
```

This will add the three numbers to the ArrayList. Now, we can iterate through the items in the ArrayList (list) using a foreach loop:

```
1    static void Main()
2    {
3        ArrayList list = new ArrayList();
4        list.Add(45);
```

```
        list.Add(87);
        list.Add(12);
        foreach(int num in list)
        {
            Console.WriteLine(num);
        }
    }
```

which will print out the elements in the ArrayList as

```
    45
    87
    12
```

The ArrayList class has also implemented the indexer property (or index operator) which allow its elements to be accessed using the [] operators, just as you do with a simple array (we will see how to implement indexers in the next lesson). The following code is similar to the above code but uses the indexers to access the elements of the ArrayList.

```
    static void Main()
    {
        ArrayList list = new ArrayList();
        list.Add(45);
        list.Add(87);
        list.Add(12);
        for(int i=0; i<list.Count; i++)
        {
            Console.WriteLine(list[i]);
        }
    }
```

The output of the code will be similar to the one presented previously. The above code uses the property Count to find the current number of elements in the ArrayList. Recall that ArrayList inherits this property (Count) from its parent interface ICollection.

A list of some other important properties and methods of the ArrayList class is presented in the following table:

Property or Method	Description
Capacity	Gets or sets the number of elements the ArrayList can contain.
Count	Gets the exact number of elements in the ArrayList.
Add(object)	Adds an element at the end of an ArrayList.
Remove(object)	Removes an element from the ArrayList.
RemoveAt(int)	Removes an element at the specified index from the ArrayList.
Insert(int, object)	Inserts an object in the ArrayList at the specified index.
Clear()	Removes all the elements from the ArrayList
Contains(object)	Returns a boolean value indicating whether the ArrayList contains the supplied element or not.
CopyTo()	Copies the elements of the ArrayList to the array supplied as a parameter. This method is overloaded and one can specify the range to be copied and also from which index of the array copy should start.
IndexOf(object)	Returns the zero based index of the first occurrence of the object in the ArrayList. If the object is not found in the ArrayList, it returns -1.
LastIndexOf(object)	Returns the zero based index of the last occurrence of the object in the ArrayList.
ToArray()	Returns an array of type object that contains all the elements of the ArrayList.
TrimToSize()	Sets the capacity to the actual number of elements in the ArrayList.

The Stack class

The System.Collections.Stack class is a kind of collection that provides controlled access to its elements. A stack works on the principle of Last In First Out (LIFO), which means that the last item inserted into the stack will be the first item to be removed from it. Stacks and Queues are very common data structures in computer science and they are implemented in both hardware and software. The insertion of an item onto the stack is termed as a 'Push' while removing an item from the stack is called a 'Pop'. If the item is not removed but only read from the top of the stack, then this is called a 'Peek' operation. The System.Collections.Stack class provides the functionality of a Stack in the .Net environment. The Stack class can be instantiated in a manner similar to the one we used for the ArrayList.

```
Stack stack = new Stack();
```

The above (default) constructor will initialize a new empty stack. The following constructor call will initialize the stack with the supplied initial capacity:

```
Stack stack = new Stack(12);
```

While the following constructor will initialize the Stack with the supplied collection:

```
Stack stack = new Stack(myCollection);
```

Now, we can push elements onto the Stack using the Push() method, like this:

```
stack.Push(2);
stack.Push(4);
stack.Push(6);
```

In a similar manner, we can retrieve elements from the stack using the Pop() method. A complete program that pushes 3 elements onto the stack and then pops them off one by one is presented below:

```
using System;
using System.Collections;
namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            Stack stack = new Stack();
            stack.Push(2);
            stack.Push(4);
            stack.Push(6);
            while(stack.Count != 0)
            {
                Console.WriteLine(stack.Pop());
            }
        }
    }
}
```

Note that we have used a while() loop here to iterate through the elements of the stack. One thing to remember in the case of a stack is that the Pop() operation not only returns the element at the top of stack, but also removes the top element so the Count value will decrease with each Pop() operation. The output of the above program will be:

```
6
4
```

```
    2
```

The other methods in the Stack class are very similar to those of an ArrayList except for the Peek() method. The Peek() method returns the top element of the stack without removing it. The following program demonstrates the use of the Peek() operation.

```
    static void Main()
    {
        Stack stack = new Stack();
        stack.Push(2);
        stack.Push(4);
        stack.Push(6);
        Console.WriteLine("The total number of elements on the stack
            before Peek() = {0}", stack.Count);
        Console.WriteLine("The top element of stack is {0}",
        stack.Peek());
        Console.WriteLine("The total number of elements on the stack
            after Peek() = {0}", stack.Count);
    }
```

The above program pushes three elements onto the stack and then peeks at the top element on the stack. The program prints the number of elements on the stack before and after the Peek() operation. The result of the program is:

```
    The total number of elements on the stack before Peek() = 3
    The top element of stack is 6
    The total number of elements on the stack after Peek() = 3
```

The output of the program shows that Peek() does not affect the number of elements on the stack and does not remove the top element, contrary to the Pop() operation.

The Queue class

A Queue works on the principle of First In First Out (FIFO), which means that the first item inserted into the queue will be the first item removed from it. To 'Enqueue' an item is to insert it into the queue, and removal of an item from the queue is termed 'Dequeue'. Like a stack, there is also a Peek operation, where the item is not removed but only read from the front of the queue.

The System.Collections.Queue class provides the functionality of queues in the .Net environment. The Queue's constructors are similar to those of the ArrayList and the Stack.

```
    // an empty queue
    Queue queue = new Queue();
    // a queue with initial capacity 16
```

```
    Queue queue = new Queue(16);
    // a queue containing elements from myCollection
    Queue queue = new Queue(myCollection);
```

The following program demonstrates the use of Queues in C#.

```
    static void Main()
    {
        Queue queue = new Queue();
        queue.Enqueue(2);
        queue.Enqueue(4);
        queue.Enqueue(6);
        while(queue.Count != 0)
        {
            Console.WriteLine(queue.Dequeue());
        }
    }
```

The program enqueues three elements into the Queue and then dequeues them using a while loop. The output of the program is:

```
    2
    4
    6
```

The output shows that the queue removes items in the order they were inserted. The other methods of a Queue are very similar to those of the ArrayList and Stack classes.

Dictionaries

Dictionaries are a kind of collection that store items in a key-value pair fashion. Each value in the collection is identified by its key. All the keys in the collection are unique and there can not be more than one key with the same name. This is similar to the English language dictionary like the Oxford Dictionary where each word (key) has its corresponding meaning (value). The two most common types of Dictionaries in the System.Collections namespace are the Hashtable and the SortedList.

The Hashtable class

Hashtable stores items as key-value pairs. Each item (or value) in the hashtable is uniquely identified by its key. A hashtable stores the key and its value as an object type. Mostly the string class is used as the key in a hashtable, but you can use any other class as a key. However, before selecting a class for the key, be sure to override the Equals() and GetHashCode() methods that it inherit from the object class such that:

- Equals() checks for instance equality rather than the default reference equality.
- GetHashCode() returns the same integer for similar instances of the class.
- The values returned by GetHashCode() are evenly distributed between the MinValue and the MaxValue of the Integer type.

Constructing a Hashtable

The string and some of the other classes provided in the Base Class Library do consider these issues, and they are very suitable for usage as a key in hashtables or other dictionaries. There are many constructors available to instantiate a hashtable. The simplest is:

```
1    Hashtable ht = new Hashtable();
```

Which is a default no argument constructor. A hashtable can also be constructed by passing in the initial capacity:

```
1    Hashtable ht = new Hashtable(20);
```

The Hashtable class also contains some other constructors which allow you to initialize the hashtable with some other collection or dictionary.

Adding items to a Hashtable

Once you have instantiated a hashtable object, then you can add items to it using its Add() method:

```
1    ht.Add("st01", "Sanfy");
2    ht.Add("sci01", "Newton");
3    ht.Add("sci02", "Einstein");
```

Retrieving items from the Hashtable

Here we have inserted three items into the hashtable along with their keys. Any particular item can be retrieved using its key:

```
1    Console.WriteLine("Size of Hashtable is {0}", ht.Count);
2    Console.WriteLine("Element with key = st01 is {0}", ht["st01"]);
3    Console.WriteLine("Size of Hashtable is {0}", ht.Count);
```

Here we have used the indexer ([] operator) to retrieve the value from the hashtable. This way of retrieval does not remove the element from the hashtable but just returns the object with the specified key. Therefore, the size before and after the retrieval operation is always same (that is 3). The output of the code above is:

```
Size of Hashtable is 3
Element with key = st01 is Sanfy
Size of Hashtable is 3
```

Removing a particular item

The elements of the hashtable can be removed by using the Remove() method which takes the key of the item to be removed as its argument.

```
static void Main()
{
    Hashtable ht = new Hashtable(20);
    ht.Add("st01", "Faraz");
    ht.Add("sci01", "Newton");
    ht.Add("sci02", "Einstein");
    Console.WriteLine("Size of Hashtable is {0}", ht.Count);
    Console.WriteLine("Removing element with key = st01");
    ht.Remove("st01");
    Console.WriteLine("Size of Hashtable is {0}", ht.Count);
}
```

The output of the program is:

```
Size of Hashtable is 3
Removing element with key = st01
Size of Hashtable is 2
```

Getting the collection of keys and values

The collection of all the keys and values in a hashtable can be retrieved using the Keys and Values property, which return an ICollection containing all the keys and values respectively. The following program iterates through all the keys and values and prints them, using a foreach loop.

```
static void Main()
{
    Hashtable ht = new Hashtable(20);
    ht.Add("st01", "Faraz");
    ht.Add("sci01", "Newton");
    ht.Add("sci02", "Einstein");
    Console.WriteLine("Printing Keys...");
    foreach(string key in ht.Keys)
    {
        Console.WriteLine(key);
    }
    Console.WriteLine("\nPrinting Values...");
```

```
        foreach(string Value in ht.Values)
        {
            Console.WriteLine(Value);
        }
    }
```

The output of the program will be:

```
    Printing Keys...
    st01
    sci02
    sci01
    Printing Values...
    Sanfy
    Einstein
    Newton
```

Checking for the existence of a particular item in a hashtable

You can use the ContainsKey() and the ContainsValue() method to find out whether a particular item with the specified key and value exists in the hashtable or not. Both the methods return a boolean value.

```
    static void Main()
    {
        Hashtable ht = new Hashtable(20);
        ht.Add("st01", "Faraz");
        ht.Add("sci01", "Newton");
        ht.Add("sci02", "Einstein");
        Console.WriteLine(ht.ContainsKey("sci01"));
        Console.WriteLine(ht.ContainsKey("st00"));
        Console.WriteLine(ht.ContainsValue("Einstein"));
    }
```

The output is:

```
    True
    False
    True
```

Indicating whether the elements in question exist in the dictionary (hashtable) or not.

The SortedList class

The sorted list class is similar to the Hashtable, the difference being that the items are sorted according to the key. One of the advantages of using a SortedList is that you can get the items in the collection using an integer index, just like you can with an array. In the

case of SortedList, if you want to use your own class as a key then, in addition to the considerations described in Hashtable, you also need to make sure that your class implements the IComparable interface.

The IComparable interface has only one method: int CompareTo(object obj). This method takes the object type argument and returns an integer representing whether the supplied object is equal to, greater than or less than the current object.

- A return value of 0 indicates that this object is equal to the supplied obj.
- A return value greater than zero indicates that this object is greater than the supplied obj
- A return value less than zero indicates that this object is less than the supplied obj.

The string class and other primitive data types provide an implementation of this interface and hence can be used as keys in a SortedList directly.

The SortedList provides similar constructors as provided by the Hashtable and the simplest one is a zero argument constructor.

```
1    SortedList sl = new SortedList();
```

The following table lists some useful properties and methods of the SortedList class

Property or Method	Description
Count	Gets the number of elements that the SortedList contains.
Keys	Returns an ICollection of all the keys in the SortedList.
Values	Returns an ICollection of all the values in the SortedList.
Add(object key, object value)	Adds an element (key-value pair) to the SortedList.
GetKey(int index)	Returns the key at the specified index.
GetByIndex(int index)	Returns the value at the specified index.
IndexOfKey(object key)	Returns the zero based index of the specified key.
IndexOfValue(object value)	Returns the zero based index of the specified value.
Remove(object key)	Removes the element with the specified key from the SortedList.
RemoveAt(int)	Removes the element at the specified index from the SortedList.
Clear()	Removes all the elements from the SortedList.
ContainsKey(object key)	Returns a boolean value indicating whether the SortedList contains an element with the supplied key.
ContainsValue(object value)	Returns a boolean value indicating whether the SortedList contains an element with the supplied value.

The following program demonstrates the use of a SortedList.

```
    static void Main()
    {
        SortedList sl = new SortedList();
        sl.Add(32, "Java");
        sl.Add(21, "C#");
        sl.Add(7, "VB.Net");
        sl.Add(49, "C++");
        Console.WriteLine("The items in the sorted order are...");
        Console.WriteLine("\t Key \t\t Value");
        Console.WriteLine("\t === \t\t =====");
        for(int i=0; i<sl.Count; i++)
        {
            Console.WriteLine("\t {0} \t\t {1}", sl.GetKey(i),
               sl.GetByIndex(i));
        }
```

```
15      }
```

The program stores the names of different programming languages (in string form) using integer keys. Then the for loop is used to retrieve the keys and values contained in the SortedList (sl). Since this is a sorted list, the items are internally stored in a sorted order and when we retrieve these names by the GetKey() or the GetByIndex() method, we get a sorted list of items. The output of the program will be:

```
1       The items in the sorted order are...
2       Key Value
3       == =====
4       7 VB.Net
5       21 C#
6       32 Java
7       49 C++
```

3.3 String Handling in C#

In C#, a string is a built in and primitive data type. The primitive data type string maps to the System.String class. The objects of the String class (or string) are immutable by nature. By immutable it means that the state of the object can not be changed by any operation. This is the reason why when we call the ToUpper() method on string, it doesn't change the original string but creates and returns a new string object that is the upper case representation of the original object. The mutable version of string in the .Net Platform is System.Text.StringBuilder class. The objects of this class are mutable, that is, their state can be changed by their operations. Hence, if you call the Append() method on a StringBuilder class object, the new string will be appended to (added to the end of) the original object. Let's now discuss the two classes one by one.

The string class and its members

We have been using the string class since our first lesson in the C# school. We have also seen some of its properties (like Length) and methods (like Equals()) in previous lessons. Here, we will describe some of the common properties and methods of the String class and then demonstrate their use in code.

Property or Method	Description
Length	Gets the number of characters the String object contains.
CompareTo(string s)	Compares this instance of the string with the supplied string s and returns an integer on the pattern of the IComparable interface.
Compare(string s1, string s2)	This is a static method and compares the two supplied strings on the pattern of the IComparable interface.
Equals(string s)	Returns true if the supplied string is exactly the same as this string, else returns false.
Concat(string s)	Returns a new string that is the concatenation (addition) of this and the supplied string s.
Insert(int index, string s)	Returns a new string by inserting the supplied string s at the specified index of this string.
Copy(string s)	This static method returns a new string that is the copy of the supplied string.
Intern(string s)	This static method returns the system's reference to the supplied string.
StartsWith(string s)	Returns true if this string starts with the supplied string s.
EndsWith(string s)	Returns true if this string ends with the supplied string s.
IndexOf(string s) IndexOf(char ch)	Returns the zero based index of the first occurrence of the supplied string s or supplied character ch. This method is overloaded and more versions are available.
LastIndexOf(string s) LastIndexOf(char ch)	Returns the zero based index of the last occurrence of the supplied string s or supplied character ch. This method is overloaded and more versions are available.
Replace(char, char) Replace(string, string)	Returns a new string by replacing all occurrences of the first char with the second char (or first string with the second string).
Split(params char[])	Identifies those substrings (in this instance) which are delimited by one or more characters specified in an array, then places the substrings into a String array and returns it.
Substring(int i1) Substring(int i2, int i3)	Retrieves a substring of this string starting from the index position i1 till the end in the first overloaded form. In the second overloaded form, it retrieves the substring starting from the index i2 and which has a length of i3 characters.

In the following program we have demonstrated the use of most of the above methods of the string class. The program is quite self-explanatory and only includes the method calls and their results.

```
using System;
namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            string s1 = "sanfy";
            string s2 = "sfy";
            string s3 = "sanfyin";
            string s4 = "C# is a great programming language!";
            string s5 = " This is the target text ";
            Console.WriteLine("Length of {0} is {1}", s1,
               s1.Length);
            Console.WriteLine("Comparision result for {0} with
               {1} is {2}", s1, s2, s1.CompareTo(s2));
            Console.WriteLine("Equality checking of {0} with {1}
               returns {2}", s1, s3, s1.Equals(s3));
            Console.WriteLine("Equality checking of {0} with
               lowercase {1} ({2}) returns {3}",
            s1, s3, s3.ToLower(), s1.Equals(s3.ToLower()));
            Console.WriteLine("The index of a in {0} is {1}", s3,
               s3.IndexOf('a'));
            Console.WriteLine("The last index of a in {0} is
               {1}", s3, s3.LastIndexOf('a'));
            Console.WriteLine("The individual words of '{0}'
               are", s4);
            string []words = s4.Split(' ');
            foreach(string word in words)
            {
                Console.WriteLine("\t {0}", word);
            }
            Console.WriteLine("\nThe substring of \n\t'{0}'
               \nfrom index 3 of length 10 is \n\t'{1}'",
            s4, s4.Substring(3, 10));

            Console.WriteLine("\nThe string \n\t'{0}'\nafter
               trimming is \n\t'{1}'", s5, s5.Trim());
        }
    }
}
```

The output of the program will certainly help you understand the behavior of each member.

```
Length of sanfy is 5
Comparison result for sanfy with snfy is -1
Equality checking of sanfy with Sanfy returns False
Equality checking of faraz with lowercase Faraz (faraz) returns
    True
The index of a in Faraz is 1
The last index of a in Faraz is 3
The individual words of 'C# is a great programming language!' are
C#
is
a
great
programming
language!
The substring of
'C# is a great programming language!'
from index 3 of length 10 is
'is a great'
The string
' This is the target text '
after trimming is
'This is the target text'
```

The StringBuilder class

The System.Text.StringBuilder class is very similar to the System.String class with the difference that it is mutable; that is, the internal state of its objects can be modified by its operations. Unlike in the string class, you must first call the constructor of a StringBuilder to instantiate its object.

```
string s = "This is held by string";
StringBuilder sb = new StringBuilder("This is held by
    StringBuilder");
```

StringBuilder is somewhat similar to ArrayList and other collections in the way that it grows automatically as the size of the string it contains changes. Hence, the Capacity of a StringBuilder may be different from its Length. Some of the more common properties and methods of the StringBuilder class are listed in the following table:

Property or Method	Description
Length	Gets the number of characters that the StringBuilder object contains.
Capacity	Gets the current capacity of the StringBuilder object.
Append()	Appends the string representation of the specified object at the end of this StringBuilder instance. The method has a number of overloaded forms.
Insert()	Inserts the string representation of the specified object at the specified index of this StringBuilder object.
Replace(char, char) Replace(string, string)	Replaces all occurrences of the first supplied character (or string) with the second supplied character (or string) in this StringBuilder object.
Remove(int st, int length)	Removes all characters from the index position st of specified length in the current StringBuilder object.
Equals(StringBuilder)	Checks the supplied StringBuilder object with this instance and returns true if both are identical; otherwise, it returns false.

The following program demonstrates the use of some of these methods:

```
using System;
using System.Text;
namespace CSharpSchool
{
    class Test
    {
        static void Main()
        {
            StringBuilder sb = new StringBuilder("The text");
            string s = " is complete";
            Console.WriteLine("Length of StringBuilder '{0}' is
                {1}", sb, sb.Length);
            Console.WriteLine("Capacity of StringBuilder '{0}' is
                {1}", sb, sb.Capacity);
            Console.WriteLine("\nStringBuilder before appending
                is '{0}'", sb);
            Console.WriteLine("StringBuilder after appending
                '{0}' is '{1}'", s, sb.Append(s));
            Console.WriteLine("\nStringBuilder after inserting
                'now' is '{0}'", sb.Insert(11, "
            now"));
            Console.WriteLine("\nStringBuilder after removing 'is
                ' is '{0}'", sb.Remove(8, 3));
            Console.WriteLine("\nStringBuilder after replacing
                all 'e' with 'x' is {0}",
```

```
                sb.Replace('e', 'x'));
            }
        }
    }
```

And the output of the program is:

```
    Length of StringBuilder 'The text' is 8
    Capacity of StringBuilder 'The text' is 16
    StringBuilder before appending is 'The text'
    StringBuilder after appending ' is complete' is 'The text is
       complete'
    StringBuilder after inserting 'now' is 'The text is now complete'
    StringBuilder after removing 'is ' is 'The text now complete'
    StringBuilder after replacing all 'e' with 'x' is Thx txxt now
       complxtx
```

Note that in all cases, the original object of the StringBuilder is getting modified, hence StringBuilder objects are mutable compared to the immutable String objects.

Postface

Keep revising.