CSC-200

Module 1

Beginning Programming with Visual C#

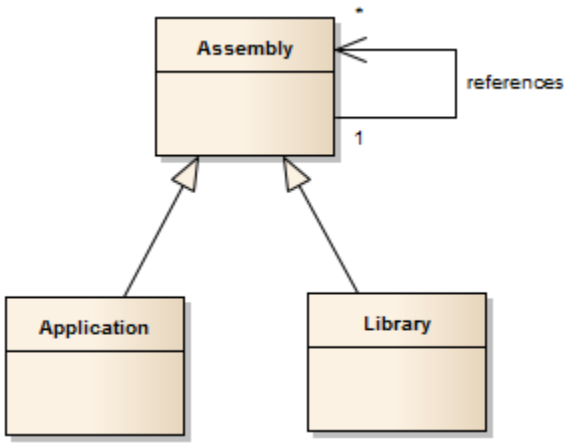
1 Assembly & Type

1.1 Assembly

An *assembly* is basically a module that contains compiled code or resources that generated by a .NET language compiler. An assembly cannot directly run on any operating system that does not have a *.NET Common Language Runtime*. If you have a correct or compatible version of a .NET CLR on a system then it can load and run assemblies.

There are two types of assembly; application assembly and library assembly. An application assembly is packaged as an executable (**.exe**) so that it can start a process to host the .NET CLR. A library assembly is a dynamic-link library (**.dll**) that can be loaded and mapped into one or more existing .NET processes but it cannot start a process. An assembly can use another assembly by referencing it during compilation.

Assembly types



Let us begin by creating two projects; one to build an application assembly and another to build a library assembly. Visual Studio does not work with projects directly without a solution. A *solution* is used to reference one or more projects that you wish to build, run and debug together as a unit. You can now create a new solution by selecting *File* | *New* | *Project…* menu option and then select the *Blank Solution* template in *Other Project Types* | *Visual Studio Solutions* group. Provide a name and location for the solution using the information shown in the next page and click the OK button to create the solution.

Solution Information

```
Solution Name    :    Module1
Solution Type    :    Visual Studio Solutions - Blank Solution
Location         :    C:\CSDEV\SRC\
```

You can view the contents for the solution using the *Solution Explorer* view that can be opened from *View* menu if it is not visible. Once the solution has been created, you can add a project to the solution to build a library assembly. Select *File* | *Add* | *Project…* menu option or right-click over the solution and then select *Add* | *New Project…* option from the shortcut menu. Use the information below for the new project.

Library project information

```
Project Name     :    MyLib1
Project Type     :    Visual C# - Windows - Class Library
Location         :    C:\CSDEV\SRC\Module1\
Solution         :    Module1
```

Once the library project has been added you can then add another one to build an application assembly instead. Select the same option again and then use the following information for the new project.

Application project information

```
Project Name     :    MyApp1
Project Type     :    Visual C# - Windows - Console Application
Location         :    C:\CSDEV\SRC\Module1\
Solution         :    Module1
```

The library project is marked as the startup project since it was the first project you added to the solution. However since you cannot launch a library directly, it should be the application project that should be marked for startup. Right-click over **MyApp1** project and select *Set As Startup Project* option.

As mentioned earlier, one assembly can use another assembly by referencing it during compilation. For **MyApp1** assembly to use the **MyLib1** assembly, add a reference to the library assembly in the application project. Right-click over the *References* folder in MyApp1 project and select the *Add Reference…* option. You can reference an assembly from different locations; the *global assembly cache* (GAC), the file system or from within the same solution. Choose *Project* page to reference an assembly from another project in the solution. Select the **MyLib1** project to add a reference to our library assembly. You can still add reference to other assemblies even if when you are not sure if you use them or not since the .NET CLR will not load in assemblies that you do not actually use in your project even though you have referenced them. But if the assembly was not referenced from the GAC, referenced assemblies would still be deployed.

You can check if an assembly will be deployed with the application by selecting the referenced assembly and use the *Properties* window to check if **Copy Local** is **True**. If the window is not visible, press **F4** or open it from the *View* menu. If you built the solution now, you can check the application output **bin** folder and see that the library assembly is also in the same folder as the application but it will not be loaded since we have not used anything from the library.

One library assembly that is always referenced and loaded is **mscorlib** even if it is not visible in the project's references. It contains core types that are required for all .NET applications and libraries to work. To know exactly which assemblies are actually loaded automatically when an application is launched you can use the .NET SDK tool named *ILDASM* to examine the internals of an assembly.

1.2 Type

An assembly usually contains at least one type. However it is still possible for a library assembly not to contain any types at all but used for storing resources instead. When you create a new library project, the project contains a *Class1.cs* source file that creates a type named **Class1**. Remove the file from the project and you can still build the library. However you cannot do the same thing for an application assembly. The project has a **Program.cs** source file that creates a class named **Program**. Remove the file from the project and you get an error if you try to compile the project or build the solution.

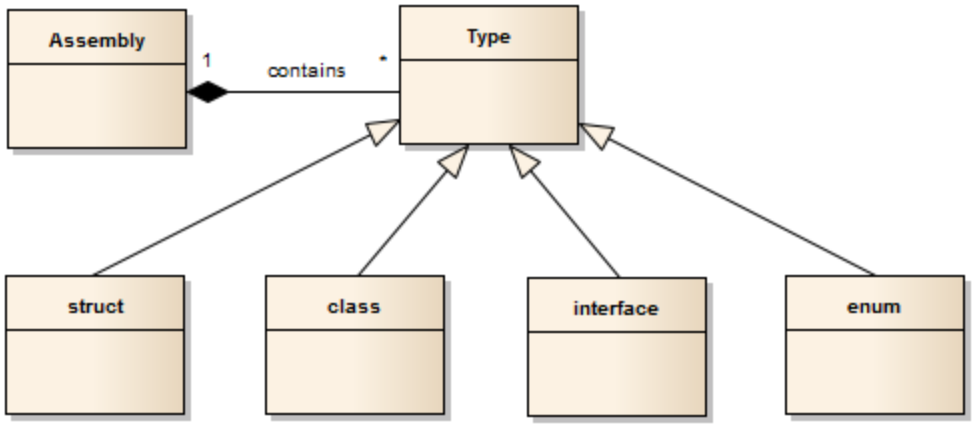
An application assembly must always have at least one type containing a **static** *Main* method. The type name does not matter as long as the method exists and has the correct signature. This method will be called automatically by .NET CLR when the application is launched so it is considered as the main entry point for applications. When the method completes and the call returns the application will then exit and the process will terminate if there are no other applications or foreground threads still running in the same process. Add back a type with a Main method so that we can compile the project again. You can use either **class** or **struct** to defined methods. Right-click on **MyApp1** and then select *Add New item…* option. Select *Code File* template to get a blank source file and name it as **Global**. Enter the following content and you should now be able to build the project.

Simplest main method: MyApp1\Global.cs

```
struct Global {
      static void Main() {
      }
}
```

Besides **class**, an assembly can contain other different types such as **interface**, **struct** and **enum**. There are also specialized types of classes such as **attribute** and **delegate**. To learn .NET programming is basically to learn how to use the existing types that is provided by *Microsoft.NET Framework* and how to create your own custom types.

Assembly & Types



A type can be referenced by using its full name. This may include a namespace if a type is defined in a **namespace**. For example **Console** type has a **System** namespace. The following shows how to call a static **WriteLine** method in the Console type to display text in the console window. We use the full type name to locate the type.

Using the System.Console type

```
class Program {
      static void Main() {
            System.Console.WriteLine("Hello!");
            System.Console.WriteLine("Goodbye!");
      }
}
```

A C# method contains a sequence of statements where the end of a statement is marked using the statement terminator (**;**) operator. A single statement can be spread over multiple lines or multiple statements can be placed into one line. Spacing is not important in C# but letter-case is crucial. C# is a case-sensitive language so lower-case and upper-case characters are not treated as equal so you must enter the above code in the exact case as shown. You can now build the project and then start the application. Use *Start without Debugging* since *Start Debugging* option closes the console window when application exits.

You can also use a **ReadLine** or **ReadKey** statement at the end of the program to stop a console window from closing until the user presses *ENTER* or another key to continue as shown in the following. Use **Write** method if you do not want to add a line-break at the end of the text so that the waiting cursor stays on the same line. You can now use *Start Debugging* to run the application.

Use input to stop console window closing

```
static void Main() {
      System.Console.WriteLine("Hello!");
      System.Console.WriteLine("Goodbye!");
      System.Console.Write("Press any key to continue...");
      System.Console.ReadKey();
}
```

Even though you are building a console application does not mean that you will not be able to use any *Graphical User-Interface* (GUI) elements in your program as it only determines whether a console window is opened or not. The problem with the console window is that it only supports one language and a single font. So you cannot display foreign characters or special characters that are not part of the selected font. If you want to be able to display any *Unicode* character, you have to use the Windows GUI.

The easiest way to display text in Windows is to use **MessageBox** type. To use this type you must reference the **System.Windows.Forms** assembly which is in the GAC, so you can add it from the *.NET* page in *Add Reference* dialog. Replace the console prompt and input with the following statement that calls a static **Show** method in MessageBox type to display text in a dialog.

Using the MessageBox type

```
static void Main() {
      System.Console.WriteLine("Hello!");
      System.Console.WriteLine("Goodbye!");
      System.Console.Write("Press any key to continue...");
      System.Console.ReadKey();
      System.Windows.Forms.MessageBox.Show("Click OK to continue.");
}
```

1.3 Namespace

Namespaces are important to resolve name conflicts when we need to use a few types from different assemblies that so happens to have the same type name. If the types are defined in different namespaces, then the namespace would be able to identify the exact type that we wish to use. However it would be tedious if we have to keep entering the same namespace each time we use a type when there is no conflict. You can then decide to import the entire namespace.

Importing all types from namespace

```
using System;
using System.Windows.Forms;
```

By importing types from their original namespace into the current namespace, it is now possible for us to reference the types directly without having to explicitly specify the namespace as shown below. This is especially useful when you have really long namespaces like **System.Windows.Forms**.

Using imported types in the current namespace

```
Console.WriteLine("Hello!");
Console.WriteLine("Goodbye!");
Console.Write("Press any key to continue...");
Console.ReadKey();
```

Even if there is a conflict, rather than having to enter the whole namespace, we can assign an *alias* to a namespace. You can then use the shorter alias instead of having to enter the entire namespace in the code.

Assigning an alias for a namespace

```
using Sys = System;
using W = System.Windows.Forms;
```

Using namespace aliases

```
Sys.Console.WriteLine("Hello!");
Sys.Console.WriteLine("Goodbye!");
Sys.Console.Write("Press any key to continue...");
Sys.Console.ReadKey();
W.MessageBox.Show("Click OK to continue.");
```

Not only can you assign an alias to a namespace but you may also assign it to a type. This may be used when a type name is too long and you need to use that type multiple times in your code or there is a conflict with the name.

Assigning an alias for a type

```
using C = System.Console;
using MsgBox = System.Windows.Forms.MessageBox;
```

Using types through aliases

```
C.WriteLine("Hello!"); C.WriteLine("Goodbye!");
C.Write("Press any key to continue..."); C.ReadKey();
MsgBox.Show("Click OK to continue.");
```

You can decide which methods serves best for you when there is name conflicts and when there isn't. Importing namespaces and using aliases can help reduce redundant repetitious usage of namespaces in your code but you should ensure that your code still remains easily readable.

You can of course declare namespaces for your types. This is especially required when building a library of types that other developers can use in their projects to reduce the chance of type name conflicts. Even though it is not so important to declare namespaces in applications since the types are mostly used internally rather publicly accessed by other assemblies. The namespace is logical and not physical so it does not have to follow the name of the assembly. For example, **System.String** type is implemented in **mscorlib** library not **System** library.

Declaring a namespace

```
namespace Symbolicon
{
      class Program {
            :
}
```

You can use as many namespaces as you like in your library or your application. Namespaces are not dedicated so multiple libraries and applications can use the same namespaces as long as they do not have types with the same name. Even though a type can exist in one namespace, the namespace may also be nested as shown below. If there is nothing declared in the outer namespace you may as well combine the namespaces into a single multi-part declaration.

Declaring nested namespaces

```
namespace Symbolicon
{
      namespace Examples
      {
                  :
```

Declaring a multi-part namespace

```
namespace Symbolicon.Examples
{
            :
```

Visual Studio can automatically generate the namespace for each type you add to the project. By default the namespace will follow the name of the project but you can change it in the *Application* page in *Project Properties* window. If you add a new type into a sub-folder in the project, Visual Studio will automatically append the folder name to the namespace for that type but since namespaces are logical rather than physical organization elements you can freely change or remove them if you wish.

1.4 Comment

Before you write more code, you should first learn how to comment your code. If you have been programming with C++ or Java, you will be familiar with C# comment operators. C# support both single-line and also delimited comments. Single-line comment must begin with the characters **//** and all content until the end of the line is ignored by the compiler. Normal compilation would continue on the next line. Delimited comment must begin with characters **/*** and ends with ***/** and all the content between the delimiters would be ignored.

Single-line comment

```
// this content will not be compiled until end-of-line
```

Delimited comment

```
/* only content within here will not be compiled */
```

You can use a special version of a single-line comment but with **///** characters that can be used to generate documentation on your types. However this is only a feature provided by Visual Studio rather than an integral part of the language. It is a feature that is recommended for public libraries that are constantly being updated so that it will be faster for the libraries to be released and consumed. It is however not recommended for applications or in-house libraries as cluttering your source code with a large amount of comments can make your code harder to read and edit. You can create external documentation and help files instead which is more preferable in this scenario when required.

2 Common Type System

2.1 Common Type System

A C# compiler does not compile source code to the native code for a particular process or platform. Instead it translates the code to a *Common Intermediate Language* (CIL) format. This will allows .NET code to run on any processor or on any platform as long as there is a .NET CLR installed. A CLR has a JIT (Just-In-Time) compiler that translates IL code to native code for each method called the first time after its assembly is loaded. The native code is then executed and will be reused for subsequent calls until the assembly is unloaded. You can also pre-generate a native code image for each assembly using the NGEN tool so that its IL does not have to be translated again each time the assembly is loaded.

All this means is that it does not matter what .NET programming language you use because all the code will be compiled to the same CIL. At this level the code can interoperate even though they were originally written in different languages and compiled using different compilers. However to ensure full interoperability they must also use the same *Common Type System* (CTS). When one method calls another method they can also pass and return objects and values. As long as the type of object and value is known to both methods then there should be no problem for them to access and process the object or value even though the type may be defined in one programming language but used by code written in a different language. Since all types are compiled to the same CTS format and its type information called as *metadata* is recorded in the assembly, referencing its assembly is all it takes to use any type.

2.2 Primitive Types

Primitive types are base IL types. You do not need to reference any assembly to use them even though additional support for them is available in **mscorlib**. You can refer to them using C# version of the types or the .NET version of the types in **System** namespace of the mscorlib library. It does not matter which one you use since they are compiled to the same primitive type. You can also use them interchangeably in your source code. The following shows the primitive types as well as how you reference them in C# and .NET and what they can store. As C# can use all .NET types you can use either version to reference them.

Mapping of primitive types in C# and .NET

C#	.NET	Size	Range of Values
byte	Byte	1	0-255
sbyte	SByte	1	-128 to +127
short	Int16	2	-32768 to +32767
int	Int32	4	-2147483648 to +2147483647
long	Int64	8	-9223372036854775808 to +9223372036854775807
ushort	UInt16	2	0 to 65535
uint	UInt32	4	0 to 4294967295
ulong	UInt64	8	0 to 18446744073709551615
float	Single	4	7 digits precision (6 decimal places)
double	Double	8	15 digits precision (6 decimal places)
bool	Boolean	1	false or true
char	Char	2	any Unicode character
decimal	Decimal	12	-+79228162514264337593543950335 (28 decimal)

Using primitive types: Types1\Program.cs

```
int   a = 123;    // using C# type
Int32 b = 321;    // using .NET type
a = b;            // int same as Int32
b = a;            // Int32 same as int
Console.WriteLine(a);
Console.WriteLine(b);
```

Primitive types are designed to store only one value. When you declare either a field or variable using a primitive type, memory is allocated accordingly to the size of the data type. The default value for these types is 0 and false. You can provide your own initialization value during declaration. Following is an example showing how to declare and initialize variables. Note that literal values for **char** must be delimited using single quotes while **float** literals must end with **f** and **decimal** literals end with **m**. Integer-based and double literals does not require any markings since they are default types. C# can automatically infer the type of a variable by what you initialize it with so you do not even have to declare the variable type but simply use **var**.

Declaring and initializing values

```
byte       v1 = 255;
char       v2 = 'A';
bool       v3 = true;
float      v4 = 9.999999f;
double     v5 = 999999999.999999;
decimal    v6 = 792281625142643.37593543950335m;
Console.WriteLine(v1); Console.WriteLine(v2);
Console.WriteLine(v3); Console.WriteLine(v4);
Console.WriteLine(v5); Console.WriteLine(v6);
```

Using C# type inference

```
var   a1 = 100;        //   int (u=uint, l=long, ul=ulong, d=double)
var   a2 = 'A';        //   char
var   a3 = 99.99f;     //   float
var   a4 = 9.9999;     //   double
var   a5 = 999.9m;     //   decimal
var   a6 = true;       //   bool
```

You can also use binary or hexadecimal numbering systems for integer values. Use **0x** prefix for hexadecimal and **0b** for binary. You can also use underscores to separate digits in long values.

Numbering systems and separating digits

```
var a7 = 0xa910d72a;              // hexadecimal value
var a8 = 0b1100111000001001;      // binary value
var a9 = 0b1100_1110_0000_1001;   // underscore to separate digits
```

2.3 Typecasting

There is built-in data conversion support between simple types. Thus there is no need to call special methods to perform explicit data conversion for these types. However, the compiler may refuse to perform certain conversions automatically if it may cause data corruption or when the data type is not compatible. In the following example, it should be no problem to transfer an **int** to a **double** since the later can store any value that you can possibly store in **int**. However, there will be a problem if we attempt to store a **double** value into an **int** since the later will not be able to store decimal places or too large a number. If you still wish the compiler to perform conversion, use type-casting. It involves placing the type to cast in between braces **( )** at the front of fields, variables and also expressions. Typecasting just enforce conversions that's not done automatically by the compiler.

Invalid automatic type conversion

```
int v1 = 100;
double v2 = v1;  // can fit
v1 = v2;         // may not fit
byte v3 = 65;
char v4 = v3;    // different type
v3 = v4;         // different type
```

Using typecasting to enforce conversion

```
int v1 = 100; double v2 = v1; v1 = (int)v2; // cast to int
byte v3 = 65; char v4 = (char)v3; // cast to char
v3 = (byte)v4; // cast to byte
```

In Microsoft .NET all objects and values can be converted to text strings. Every type has a **ToString** method that can be called to convert the current value and object to a string. **String** class also provides a static **Format** method that can be called to format and convert one or more values into a single string object by providing a format string containing formatting codes. You can also use a **$** string operator to interpolate variables and expressions directly into the format string. This is just a simplified way to call **String.Format** method.

Unformatted & formatted strings

```
double amount = 32500.90;
string text1 = amount.ToString();
string text2 = amount.ToString("###,##0.00");
string text3 = String.Format("{0:###,##0.00}",amount);
string text4 = $"{amount:###,##0.00}"; // same as String.Format()
Console.WriteLine(text1); Console.WriteLine(text2);
Console.WriteLine(text3); Console.WriteLine(text4);
```

All simple types have a **Parse** method that we can call to convert a text string that type. In the following example, we demonstrate how to read text strings from the console using **ReadLine** method and convert them to **int** and **decimal** types using parse methods provided. Note that exceptions can be thrown if the **Parse** method fails. You can use a **TryParse** method that will not generate an exception but returns **false** instead.

Parsing strings

```
Console.Write("Enter account id:");
int id = Int32.Parse(Console.ReadLine());
Console.Write("Enter amount to deposit:");
decimal amount = Decimal.Parse(Console.ReadLine());
Console.WriteLine(id); Console.WriteLine(amount);
```

You can also call the methods in the **Convert** class to convert between simple types. To simplify conversion from strings, Microsoft.NET groups all conversions into one **Convert** class. You can then call the methods provided by in the class rather than using the parse method of each type directly.

Using methods in the Convert class

```
Console.Write("Enter account id:");
id = Convert.ToInt32(Console.ReadLine());
Console.Write("Enter amount to deposit:");
amount = Convert.ToDecimal(Console.ReadLine());
Console.WriteLine(id); Console.WriteLine(amount);
```

2.4 Value & Reference Types

In Microsoft.NET there are two distinct category of types; reference types and value types. It affects how memory is allocated for fields and variables declared for a type. Memory will be allocated in-place for value types. It means that the content is stored in the same location where a field or variable is defined. When you access a field or variable of a value type, you are also accessing the content directly. For reference types, content is stored into a separate memory location allocated from a dynamic heap referred to as managed memory. The reference to the memory location is stored in the field or variable instead. The reference will be required to locate and fetch the content from managed memory. Things that are stored in managed memory are commonly called objects. Use the term instance to refer to memory allocated to store data regardless of whether its for value or reference type.

To understand differences between using value and reference types clearly, you should first know how to create these types. In C# use the **struct** keyword to create a value type and **class** keyword to create a reference type. The following example shows these two types.

Creating a value type: MyLib1\Point1.cs

```
public struct Point1 {
      public int X;
      public int Y;
}
```

Create a reference type: MyLib1\Point2.cs

```
public class Point2 {
      public int X;
      public int Y;
}
```

Using a value type is straightforward. When we declare a variable, the memory that is allocated for the variable will be used to store **X** and **Y**, which represent the value but when you declare a variable for a reference type, memory is allocated for the variable but this memory is not used to store the value. It is used to store the reference to another memory location where the value will be stored. While the use of **new** is optional for a value type it is compulsory to use it for reference type to get back a reference to store. Using **new** on value types will simply reset the fields back to default values.

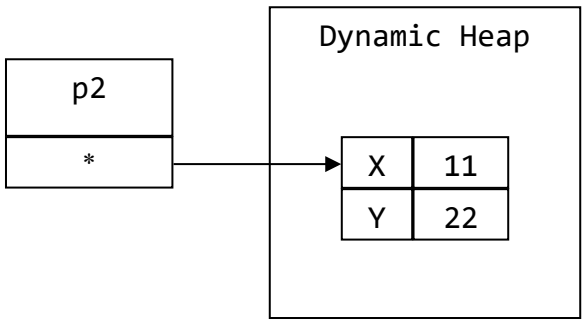
Using a value type

```
Point1 p1;       //   allocate memory to store x and y
p1.X = 10;       //   store x into p1
p1.Y = 20;       //   store y into p1
```

p1	
X	10
Y	20

Using a reference type

```
Point2 p2; // field to store reference
p2 = new Point2();    // create object
p2.X = 11;       // store x in object
p2.Y = 22;       // store y in object
```



For a value type, when a variable is destroyed, the value will also be destroyed at the same time. However for a reference type, when the variable is destroyed only the reference is destroyed. The object is still in managed memory and is only destroyed when the garbage collector runs. Even though we do not destroy the value, we should still assign **null** to the variable to release the value when we no longer wish to use it. The garbage collector can then destroy the object earlier even though the variable still exists.

Releasing an object earlier

```
p2 = null;
```

There are certain methods that accept only reference types but not value types. This is not a problem in Microsoft .NET since you can easily convert a value to an object by using **object** type. When you attempt to assign a value type to a field or variable typed **object**, a boxing operation will occur when an object is created in the heap and the value will be stored in the object. When you assign it to a value typed variable, the actual value is extracted from inside the object. This operation is called as unboxing. Typecasting is required when extracting value inside an object since it can store any value and the type of value cannot be determined at compilation time.

The following example shows when boxing and unboxing operations will occur. Note that strings are not boxed and unboxed because they are not value types. String is implemented in Microsoft.NET as a reference type thus we can say that all text strings are objects. However you still need to perform typecasting to cast an object reference back to a **string** object reference but this is only just for type-checking.

Boxing and un-boxing values from objects

```
object o1 = "123";          //    Implicit casting to string
object o2 = 123;            //    Boxing int inside an object
string s1 = (string)o1;     //    Check type of object is string
int v1 = (int)o2;           //    Unboxing an int from inside an object
```

2.5 String

String type is actually a class so strings are objects. However it is a type that is directly understood and supported so languages can provide a simpler syntax of creating and using strings. However each literal string is considered an object which is then assign to a variable or field. Assigning a new string to will release the previous one. A string has a number of methods for extracting, searching and processing the characters of the string. Note that methods that you call do not modify the existing string but will create and return a new string instead. If you wish to use the same variable, assign the new string back to the variable and release the old string.

Strings are immutable: String1\Program.cs

```
string s1 = "ABCDEF";            // assign a new string object
string s2 = s1.Insert(1,"XYZ");  // return new string so s1 not changed
Console.WriteLine(s2); Console.WriteLine(s1);
s1 = s1.Insert(1,"XYZ");         // s1 now points to new string
Console.WriteLine(s1);           // previous string reference is released
```

You can convert text in a string to uppercase or lowercase using the **ToUpper** and **ToLower** methods. To find length of a string use the **Length** property. You can insert characters into a text string by using **Insert**, **PadLeft** or **PadRight**. You can extract characters using **SubString** and remove characters by using **Remove**, **Replace** and **Trim**. Compare or search strings using **CompareTo**, **IndexOf**, **StartsWith** and **EndsWith**. A static **Format** method can be used to construct and format a text string using a set of parameters.

Using string methods

```
s1 = "   Can of Coke   ";
Console.WriteLine(s1.Length);
Console.WriteLine(s1.ToUpper());
Console.WriteLine(s1.ToLower());
Console.WriteLine(s1.PadLeft(30));
Console.WriteLine(s1.PadLeft(30,'*'));
Console.WriteLine(s1.PadRight(30,'*'));
Console.WriteLine(s1.TrimStart().TrimEnd());
Console.WriteLine(s1.Replace("Coke","Pepsi"));
Console.WriteLine(s1.Trim().Substring(0,3));
Console.WriteLine(s1.Trim().Substring(4));
Console.WriteLine(s1.IndexOf("of"));
Console.WriteLine(s1.LastIndexOf('o'));
Console.WriteLine(s1.Contains("Coke"));
Console.WriteLine(s1.Trim().StartsWith("Can"));
Console.WriteLine(s1.Trim().EndsWith("Coke"));
Console.WriteLine(s1.Remove(4,3));
Console.WriteLine(s1.Trim().Equals("Can of Coke"));
```

```
Console.WriteLine("Anne".CompareTo("Mary"));
Console.WriteLine("Anne".CompareTo("Anna"));
Console.WriteLine("Anne".CompareTo("Anne"));
string s2 = String.Format("Price of {0} is {1}",s1,1.20);
Console.WriteLine(s2);
string s3 = "This is a string.";
```

A string can be regarded as a collection of characters. A **foreach** keyword in C# is used to retrieve each item from an array or collection and repeat a statement or a block of statements enclosed in a code block ({ }). Each item is assigned to a variable before processing the statement or code block.

Accessing items from an array or collection

```
foreach (char c in s3) Console.WriteLine(c);
```

You can construct a string from multiple parts where each part may be any type by using the **+** string concatenation operator. Each is automatically converted to a string by calling its **ToString** method and combined to form a new string that is returned as the result. The compiler would automatically translate the operation into a single call to a static **Concat** method from the **String** class but the method does not support formatting. To format you may prefer the **Format** method of the **String** class. Internally all methods uses a **StringBuilder** class from **System.Text** namespace to help construct a string. Once construction has completed the **ToString** method has to be called to obtain the string. The class supports formatting and non-formatting methods to insert, delete, and replace characters and not only just for concatenation.

Building a string

```
int id = 100;
string name = "Can of Coke";
double price = 1.20;
string s1 = "ID="+id+",Name="+name+",Price="+price;
string s2 = String.Concat("ID=",id,",Name=",name,",Price=",price);
string s3 = String.Format("ID={0},Name={1},Price={2:C2}",
     id,name,price);
string s4 = $"ID={id},Name={name},Price={price:C2}"
Console.WriteLine(s1); Console.WriteLine(s2);
Console.WriteLine(s3); Console.WriteLine(s4);

StringBuilder sb = new StringBuilder();
sb.Append("ID=").Append(id).Append(",Name=").Append(name).
AppendFormat(",Price={0:C}",price);
string s4 = sb.ToString();
Console.WriteLine(s4);
sb = null;
```

You can use **Regex** class from **System.Text.RegularExpressions** namespace to compile a regular expression to perform complex validation and extraction on text. In the following example, we wish to validate user input to ensure that the user has entered a valid zip code that contains 5 digits. Call **IsMatch** method to validate text against a regular expression. A **do while** loop will be used to keep prompting the user until input data is valid.

Compiling a regular expression: Regex1\Program.cs

```
string zip = null;
bool zipIsValid = false;
Regex rx = new Regex(@"^\d{5}$");
```

Matching text against a regular expression

```
do {
      Console.Write("Enter zip code: ");
      zip = Console.ReadLine(); zipIsValid = rx.IsMatch(zip);
      if (!zipIsValid) Console.WriteLine("Invalid zip code.");
}     while (!zipIsValid);
```

Data can be extracted from matching data based on groups in an expression. In the following example, we want to ensure that the user enters a valid date but we also want to extract the day, month and year information from the input. Call **Match** method to obtain a **Match** object which not only stores the result of the matching but the data retrieved from extracting groups in the regular expression.

Use **Success** property to determine that a match is successful before accessing the result. **Value** and the item at index 0 of the **Groups** collection will return the entire text matched. Indexes from 1 onwards represent groups that were specifically present in the regular expressions.

Extracting matching groups

```
Match m = null;
Regex rx = new Regex(@"^(\d{2})/(\d{2})/(\d{4})$");
do {  Console.Write("Enter date (dd/mm/yyyy): ");
      string text = Console.ReadLine(); m = rx.Match(text);
}     while (!m.Success);
```

Displaying group values

```
Console.WriteLine(m.Value);
Console.WriteLine(m.Groups[0].Value);
Console.WriteLine(m.Groups[1].Value);
Console.WriteLine(m.Groups[2].Value);
Console.WriteLine(m.Groups[3].Value);
```

Sometimes we expect that a string may contain multiple instances of matching text and we need to match and access each instance. You can call the **Matches** method to return **MatchCollection** that contains zero or more **Match** objects. Use **foreach** to retrieve and process each **Match** object. **Replace** method can be used to replace the matching text or use **Split** to extract items to an array.

Matching multiple items

```
Regex rx = new Regex(@"\d+");
string text = "1,99,123,77,2182,22,5";
MatchCollection mc = rx.Matches(text);
if (mc.Count == 0) { Console.WriteLine("no matches found."); return; }
foreach (Match m in mc) { Console.WriteLine(m.Value); }
Console.WriteLine(rx.Replace(text, "0"));
string [] values = new Regex(@",").Split(text);
```

2.6 DateTime

Even though **DateTime** is not exposed as a C# type it is still commonly used to store, extract and manipulate date and time information. While this type is used to represent a specific date and time, there is a **TimeSpan** type that is used to represent the duration of time. When constructing a new date, you can provide a year, month and day for date information and an hour, minute, second and millisecond for time information. You can get the current date and time using a **Now** static property or just the date by using a **Today** static property. It is possible to obtain a universal date and time by using **UTCNow** property.

Construct & retrieve datetime: DateTime1\Program.cs

```
DateTime d1 = new DateTime();
DateTime d2 = new DateTime(2002,10,16);
DateTime d3 = new DateTime(2002,10,16,18,30,25);
DateTime d4 = DateTime.Now;
DateTime d5 = DateTime.Today;
DateTime d6 = DateTime.UtcNow;
Console.WriteLine(d1); Console.WriteLine(d2);
Console.WriteLine(d3); Console.WriteLine(d4);
Console.WriteLine(d5); Console.WriteLine(d6);
```

You can extract parts of a DateTime using a range of properties and methods as demonstrated below. You can convert between local and universal time and also perform calculations using **Add** and **Subtract** methods. TimeSpan can be used to represent the difference between two dates and adding or subtracting from DateTime as well. If you wish to get month names and not numeric values, you can always use **Format** method that will always give you the names base on the current culture. Culture represents language and country.

Retrieve datetime parts

```
Console.WriteLine(d4.Day); Console.WriteLine(d4.Month);
Console.WriteLine(d4.Year); Console.WriteLine(d4.DayOfWeek);
Console.WriteLine((int)d4.DayOfWeek); Console.WriteLine(d4.DayOfYear);
Console.WriteLine(d4.Hour); Console.WriteLine(d4.Minute);
Console.WriteLine(d4.Second); Console.WriteLine(d4.Millisecond);
Console.WriteLine(DateTime.DaysInMonth(d4.Year,d4.Month));
Console.WriteLine(DateTime.IsLeapYear(d4.Year));
Console.WriteLine(string.Format("{0:MMMM}",d4));
```

Calculate datetime

```
DateTime d1 = DateTime.Now;
DateTime d2 = d1.ToUniversalTime(); DateTime d3 = d2.ToLocalTime();
Console.WriteLine(d1); Console.WriteLine(d2); Console.WriteLine(d3);
Console.WriteLine(d1.AddDays(7)); Console.WriteLine(d1.AddDays(-7));
Console.WriteLine(d1.AddMonths(3)); Console.WriteLine(d1.AddYears(2));
Console.WriteLine(d1.AddHours(1)); Console.WriteLine(d1.AddMinutes(10));
Console.WriteLine(d1.AddSeconds(20));
Console.WriteLine(d1.AddMilliseconds(50));
DateTime d4 = new DateTime(1967,10,6,6,0,0);
TimeSpan t1 = new TimeSpan(1,2,3,4);
TimeSpan t2 = d1.Subtract(d4);
Console.WriteLine(d1.Add(t1));
Console.WriteLine(d1.Subtract(t1));
Console.WriteLine(t2.Days); Console.WriteLine(t2.Hours);
Console.WriteLine(t2.Minutes); Console.WriteLine(t2.Seconds);
```

Converting entire TimeSpan to a single value:

```
Console.WriteLine(t2.TotalDays); Console.WriteLine(t2.TotalHours);
Console.WriteLine(t2.TotalMinutes); Console.WriteLine(t2.TotalSeconds);
```

Date conversion

```
string s1 = "Oct 6 1967"; d1 = DateTime.Parse(s1);
Console.WriteLine(d1.ToShortDateString());
Console.WriteLine(d1.ToShortTimeString());
Console.WriteLine(d1.ToLongDateString());
Console.WriteLine(d1.ToLongTimeString());
Console.WriteLine("{0:ddd dd MMMM yyyy}",d1);
```

Methods that formats and parses numbers and dates can accept a **CultureInfo** object. This allows you to support multiple formats in one application. You can retrieve the current culture from **Thread.CurrentCulture** property or retrieve a specific culture by calling **CultureInfo.CreateSpecificCulture** and supplying the culture code. A culture code consists of an two-letter ISO language code optionally followed by a two-letter ISO country code separated by a dash.

Example culture code

```
fr-CA       Language=French, Country=Canada
```

Using multiple cultures for formatting

```
CultureInfo c1 = CultureInfo.CreateSpecificCulture("fr");
CultureInfo c2 = CultureInfo.CreateSpecificCulture("ms-MY");
Console.WriteLine(string.Format(c1, "{0:ddd dd MMMM yyyy}", d1));
Console.WriteLine(string.Format(c2, "{0:ddd dd MMMM yyyy}", d1));
```

Parsing using a specific culture

```
s1 = "6 Oktober 1967";
Console.WriteLine(DateTime.Parse(s1, c2));
```

3 Arrays

3.1 Creating Arrays

An array is a type of object that is specially handled. Just like any reference type, no space is allocated when a field or variable is declared as an array using square braces that represent the indexer operator. You need to use the **new** keyword to physically allocate space for an array. We can use an index with the indexer operator to access back an element in the array. Index for a Microsoft .NET array always begins at **0**. When you do not wish to use an array anymore, you can release it.

Creating and accessing an array: Array1\Program.cs

```
int[] a1;
a1 = new int[4];
a1[0] = 10;
a1[1] = 20;
a1[2] = 30;
a1[3] = 40;
Console.WriteLine(a1[0]);
Console.WriteLine(a1[3]);
a1 = null;
```

You can initialize an array the moment you allocate it. You can also combine the declaration, allocation and initialization into a single statement.

Initializing a new array

```
int[] a2;
a2 = new int[4] { 10, 20, 30, 40 };
a2 = null;
```

Array declaration, allocation and initialization

```
int[] a3 = { 10, 20, 30, 40 };
```

You can create multi-dimensional arrays by using extra indexes to identify the number of dimensions. When initializing such an array you need to use multiple levels of curly braces to identify all the elements to go into each dimension of the array. The following example shows how to declare, allocate and initialize such arrays in your code.

Initializing a multi-dimensional array

```
int [,] a = {
      {10,20,30},
      {30,40,50},
      {50,60,70},
      {70,80,90}
};

int [,] b;
b = new int [4,3] {
      {10,20,30},
      {30,40,50},
      {50,60,70},
      {70,80,90}
};

Console.WriteLine(a[0,0]);
Console.WriteLine(b[3,2]);
```

You can also create arrays that contain inner arrays. Since each inner array will be separately allocated, they can also have a different size which is not possible with multi-dimensional array. You will need to allocate space for the top array before allocating for inner arrays as shown in the example below. This kind of arrays is commonly called as jagged arrays.

Creating and initializing jagged arrays

```
int [][] a; a = new int [3][];
a[0] = new int [3] {10,20,30};
a[1] = new int [2] {30,40};
a[2] = new int [4] {40,50,60,70 };
Console.WriteLine(a[0][0]);
Console.WriteLine(a[2][3]);
```

Allocation and initializing jagged arrays in one statement

```
int [][] b;
b = new int [3][] {
      new int [3] {10,20,30 },
      new int [2] {30,40 },
      new int [4] {40,50,60,70 }};
```

Use **Length** property to find out the number of elements in an array regardless of how many dimensions the array has. To locate this information for each dimension, use **GetLength** method instead. Use **Rank** to find out the number of dimensions. To get upper bound index for each of the dimensions, call the **GetUpperBound** method. There is also a **GetLowerBound** method but this is not usually used since it always returns **0**.

Checking size of an array

```
int [,] p = {{100,160},{220,140},{115,270}};
Console.WriteLine(p.Length);
Console.WriteLine(p.GetLength(0));     // 1st dimension
Console.WriteLine(p.GetLength(1));     // 2nd dimension
Console.WriteLine(p.Rank);             // dimensions
Console.WriteLine(p.GetUpperBound(0));
Console.WriteLine(p.GetUpperBound(1));
```

3.2 The Array Class

Since arrays are objects, there is an **Array** class to manage arrays. However you do not use this class directly for creating array objects but it still provides a set of additional features that you can apply to all arrays such as checking the array size, copying and clearing the contents of the array as well as sorting and searching. The following demonstrate these properties and methods.

To sort and search an array, you need to use **Sort** and **BinarySearch** methods. Note that the array needs to be sorted before performing a search. A successful search will return a positive number indicating index of the element found while a failed search returns a negative number indicating the closest index in the array where the element could not be found. Both are static methods.

Sorting and searching

```
string [] a = {"John","Anne","Mary","Lilo"};
Array.Sort(a);
Console.WriteLine(a[0]); Console.WriteLine(a[3]);
int pos1 = Array.BinarySearch(a,"Lilo");
int pos2 = Array.BinarySearch(a,"Jack");
Console.WriteLine(pos1); Console.WriteLine(pos2);
```

To make an entire copy of an array, use the **Clone** method. Since an array is an object, assigning an array variable from one variable to another does not make a copy of the array since you are only copying the reference and not the value. The **Clone** method is available on certain objects to make a copy of the object. Note that the method will return the array as **object** and thus type-casting is required for assignment.

Cloning an array

```
string [] b = a;                       // copy the reference
string [] c = (string[])a.Clone();     // clone the array
a[3] = "Stitch";
Console.WriteLine(b[3]);    // value changed
Console.WriteLine(c[3]);    // value not affected
```

To copy parts of an array, use **Copy** or **CopyTo** rather than using **Clone**. The **Copy** method can copy a number of elements from the start of array to another array while **CopyTo** copy elements from start of an array to insert into any position in another array. You may use **Clear** to reset the values in the array. For example, clearing an **int** array will set all of the elements in the array to **0**. You can specify the start index and the number of elements to clear.

Copying and clearing an array

```
b = new string [3]; c = new string [5];
Array.Copy(a,b,3); a.CopyTo(c,1); Array.Clear(a,0,4);
Console.WriteLine(b[0]); Console.WriteLine(c[1]);
Console.WriteLine(a[2]);
```

3.3 Accessing Arrays

You may use **for** or **foreach** to access elements from an entire array one by one. A **foreach** will treat an entire array as one single dimension while you may use **for** to access multi-dimensional or jagged arrays.

Accessing arrays

```
int [,] values = {{1,2,3},{3,4,5},{5,6,7},{8,9,0}};
foreach (int value in values) Console.WriteLine(value);
int my = values.GetUpperBound(0);
int mx = values.GetUpperBound(1);
for (int y = 0;y <= my;y++)
     for (int x = 0;x <= mx;x++) { int value = values[y,x];
          Console.WriteLine(value); }}
```

3.4 Binary Conversion

The **BitConverter** class is used for converting values to binary and vice-versa. Binary data will be stored in memory as a byte array. In the following examples we show the methods to call to convert values to binary and from binary back to values.

Converting values to binary: Binary1\Program.cs

```
int v1 = 123;
double v2 = 99.1;
bool v3 = false;
byte[] a1 = BitConverter.GetBytes(v1);
byte[] a2 = BitConverter.GetBytes(v2);
byte[] a3 = BitConverter.GetBytes(v3);
Console.WriteLine(a1.Length);
Console.WriteLine(a2.Length);
Console.WriteLine(a3.Length);
```

Converting binary to values

```
Console.WriteLine(BitConverter.ToInt32(a1,0));
Console.WriteLine(BitConverter.ToDouble(a2,0));
Console.WriteLine(BitConverter.ToBoolean(a3,0));
```

You should use **Encoding** class when converting text to binary and vice-versa as you can decide to use ASCII or Unicode. ASCII will encode each character to a byte but this only works for English Latin characters and all foreign characters will be corrupted. Unicode can encode foreign characters but each character will take up 2 bytes. Alternatively you can use the UTF8 encoding which can encode English Latin characters to 1 byte and foreign characters to 2 bytes. UTF8 is recommended if you are unsure about the content to be encoded.

Encoding text to binary

```
string text = "This is a text string!";
byte[] a1 = Encoding.ASCII.GetBytes(text);
byte[] a2 = Encoding.Unicode.GetBytes(text);
byte[] a3 = Encoding.UTF8.GetBytes(text);
Console.WriteLine(a1.Length);
Console.WriteLine(a2.Length);
Console.WriteLine(a3.Length);
```

Decoding binary to text

```
Console.WriteLine(Encoding.ASCII.GetString(a1));
Console.WriteLine(Encoding.Unicode.GetString(a2));
Console.WriteLine(Encoding.UTF8.GetString(a3));
```

3.5 Parameter Arrays

You can pass anything as a parameter including arrays. The following example shows a method can adds up all the values of an array passed as a parameter and return the total.

Passing arrays in parameters: Array2

```
static double Sum(double[] values) {
      double total = 0;
      foreach (double value in values) total += value;
      return total;
}

static void Main() {
      double[] v1 = { 1.1, 2.5, 6.2, 9.7 };
      Console.WriteLine(Sum(v1));
}
```

You can assign a **params** keyword to any array parameter to allow the caller to pass the array as individual parameters called as a parameter array. Following shows how to pass variable number of values to a method. The method can still accept arrays but can also accept each value as a separate parameter. If there are additional parameters for the method, the parameter array must always be the last parameter.

Implementing a parameter array

```
static double Sum(params double[] values) {
      :
}

static void Main() {
      double[] v1 = { 1.1, 2.5, 6.2, 9.7 };
      Console.WriteLine(Sum(v1));
      Console.WriteLine(Sum());              // empty array
      Console.WriteLine(Sum(1.1));           // array with 1 element
      Console.WriteLine(Sum(1.1, 2.5));      // array with 2 elements
      Console.WriteLine(Sum(1.1, 2.5, 6.2, 9.7)); // 4 elements
}
```

4 Collections

4.1 ArrayList Class

Collections are types used for storing a list of objects. You can use collections by importing **System.Collections** namespace. **ArrayList** is an object similar to an array but is more dynamic since you can add and remove items from the array list at any time. Sorting and searching features are also supported. Use **Add** to add an item at the back of the list. You may also use **Insert** to add an item to any location within the list. Use the index operator to retrieve the item at any position and you can use the **Remove** or **RemoveAt** to remove an item from anywhere in the list.

Using an arraylist: Collect1\Program.cs

```
ArrayList a = new ArrayList();
a.Add("John");
a.Add("Jane");
a.Add("Mary");
a.Add("Lilo");
a.Add("Anne");
Console.WriteLine(a.Count);
Console.WriteLine(a[0]);
Console.WriteLine(a[4]);
a.Sort();
Console.WriteLine(a[0]);
Console.WriteLine(a[4]);
Console.WriteLine(a.Contains("John"));
Console.WriteLine(a.IndexOf("Lilo"));
Console.WriteLine(a.BinarySearch("Jane"));
a.Remove("Anne"); // remove by value
Console.WriteLine(a[0]);
a.RemoveAt(0); // remove by position
Console.WriteLine(a[0]);
```

4.2 HashTable Class

A hash table is very useful to store a list of items that you need to locate very quickly. You need to assign a key to each item stored. The key can then be used to retrieve or access items stored by using the array indexer operator with the key. You may also obtain the array of keys by using the **Keys** property and an array of values using the **Values** property.

Managing a hash: Collect2\Program.cs

```
Hashtable t = new Hashtable();
t.Add("John","john_lee@axn.com");
t.Add("Anne","anne688@hotmail.com");
t.Add("Mary","mary22@yahoo.com");
Console.WriteLine(t.Count);
Console.WriteLine(t["John"]);
Console.WriteLine(t["Anne"]);
Console.WriteLine(t["Mary"]);
//    Retrieve all values
foreach(string Value in t.Values)
Console.WriteLine(Value);
//    Retrieve all keys and values
foreach(string Key in t.Keys) {
      string Value = (string)t[Key];
      Console.WriteLine("{0} -> {1}",Key,Value); }
Console.WriteLine(t.ContainsKey("John"));
```

4.3 Using Generic Collections

Collections are specially designed to store objects. However when values are added to a collection, boxing operations occur. When values are accessed from collections, un-boxing operations will occur. Even if you are accessing objects, we would still need to type-cast. Examine the following program that uses the **ArrayList** collection.

Boxing, un-boxing and casting required for collections: Collect3\Program.cs

```
using System;
using System.Collections;

class Program {
      static void Main() {
            ArrayList list = new ArrayList();

            list.Add(10);         // boxing
            list.Add(20);         // boxing
            list.Add(30);         // boxing

            var v1 = (int)list[0];      // unboxing
            var v2 = (int)list[1];      // unboxing
            var v3 = (int)list[2];      // unboxing

            Console.WriteLine(v1);
            Console.WriteLine(v2);
            Console.WriteLine(v3);
      }
}
```

You now have access to generic collections since C# 2.0. When you use generic collections, you can specify type of data that you wish to put in the collection. Thus there is no boxing and un-boxing operation required as the compiler would use the exact type specified. In the following example, a generic collection class named **List** is used to replace the non-generic **ArrayList**.

Using a generic collection

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

class Program  {
      static void Main() {
            List<int> list = new List<int>();

            list.Add(10);         // no boxing
            list.Add(20);         // no boxing
            list.Add(30);         // no boxing

            int v1 = list[0];     // no unboxing
            int v2 = list[1];     // no unboxing
            int v3 = list[2];     // no unboxing

            Console.WriteLine(v1);
            Console.WriteLine(v2);
            Console.WriteLine(v3);
      }
}
```

Instead of using **Hashtable**, you can also use the generic **Dictionary** class. To use this collection, you need to specify both the type of the key and the type of the value. In the following example, both the key and value are strings.

Using Dictionary generic collection class

```
Dictionary<string, string> t =
      new Dictionary<string, string>();

t.Add("John", "john_lee@axn.com");
t.Add("Anne", "anne688@hotmail.com");
t.Add("Mary", "mary22@yahoo.com");
            :
```

There are also other more specialized collection types available such as **Stack** and **Queue** for FILO or FIFO. Stack contains **Push** and **Pop** methods to add and remove items and Queue uses **Enqueue** and **Dequeue** methods instead.

Using stacks and queues

```
Stack<int> s = new Stack<int>();
Queue<int> q = new Queue<int>();
s.Push(100);
s.Push(200);
s.Push(300);
q.Enqueue(100);
q.Enqueue(200);
q.Enqueue(300);
Console.WriteLine(s.Count);
Console.WriteLine(q.Count);
Console.WriteLine(s.Peek());
Console.WriteLine(q.Peek());
while (s.Count > 0) Console.WriteLine(s.Pop());
while (q.Count > 0) Console.WriteLine(q.Dequeue());
```