Net Centric Computing

CSC367 (Theory + Lab)

Language Preliminaries

Introduction to .NET Framework

.NET Framework: The .NET Framework is a platform for building, deploying, and running Services and applications. It is a software framework developed by Microsoft that runs primarily on Microsoft Windows. It includes a large class library called Framework Class Library (FCL).

.NET is a free, cross-platform open source developer platform for building many different types of applications.

Cross-Platform : Able to be used with different types of computer system.

With .NET, we can use multiple languages, editors, and libraries to build for web, mobile, desktop, games, AI and IoT. The Framework provides an execution environment, simplified development , deployment, and integration with a variety of programming languages, including Visual Basic, F# and C#.

The two major components of .NET Framework are the **Common Language Runtime (CLR)** and the .NET **Framework Class Library (FCL)**. The CLR is the execution engine that handles running applications. The FCL provides a set of APIs and types for common functionality.It is built on top of the CLR and provides services needed by modern applications.

Key components of .NET Framework

Common Language Runtime (CLR)

.NET Framework Class Library (FCL)

Common Type System

Common Language Specification

Application Domains

Runtime Host

Metadata and Self-Describing Components

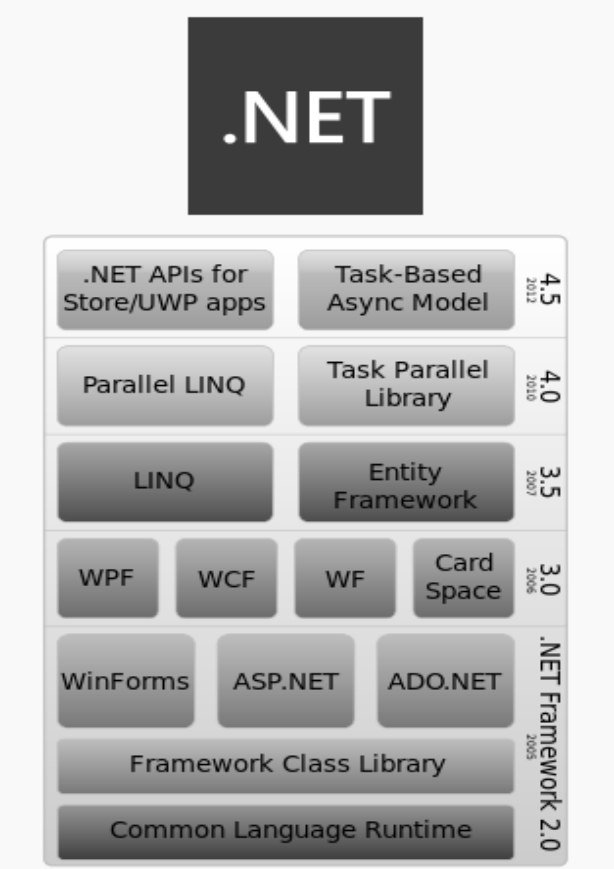
Cross-Language Interoperability

.NET Framework Security

Profiling

Side-by-Side Execution

.NET Framework Component Stack



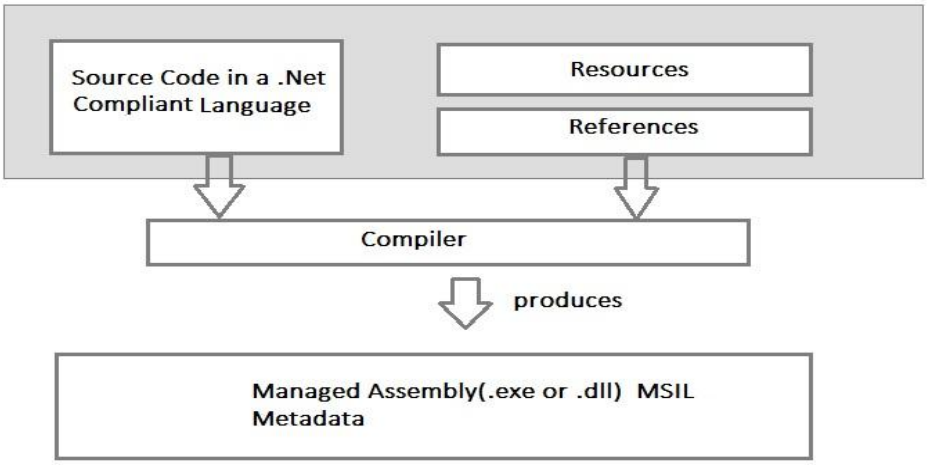
.NET Framework component stack

Developer(s)	Microsoft
Initial release	February 14, 2002; 18 years ago
Stable release	4.8.0 Build 3928 / July 25, 2019; 12 months ago[1]
Operating system	Windows 98 or later, Windows NT 4.0 or later
Type	Software framework
License	Mixed; see § Licensing
Website	www.microsoft.com/net

Overview of Compilation and Execution in .NET Framework

Compilation process

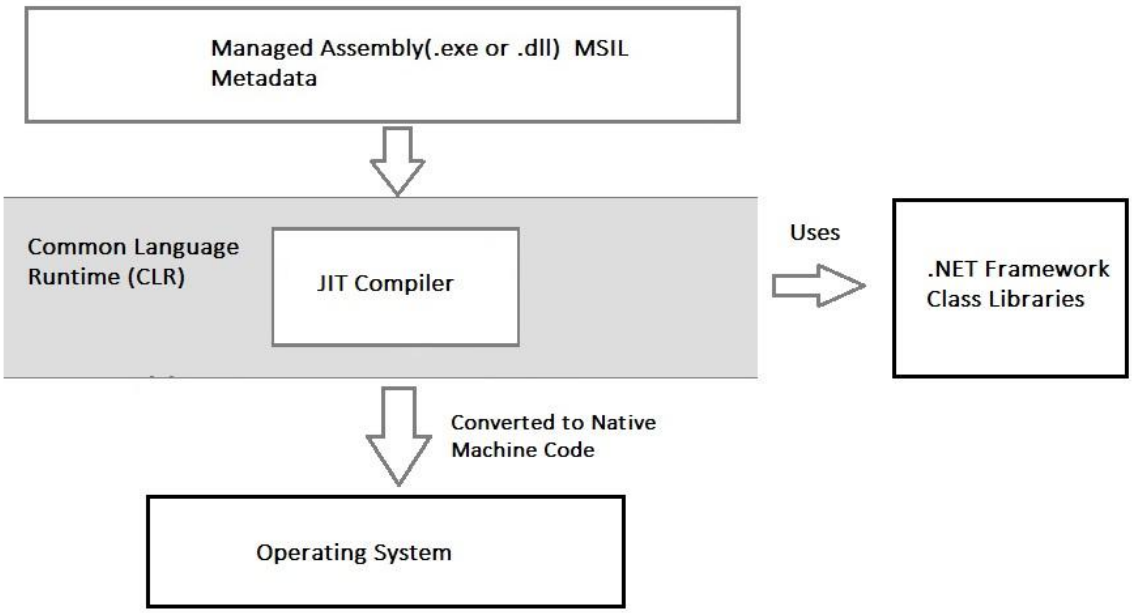
The compiler for a .NET language takes a source code file and produces an output file called an assembly. An assembly is either an executable or a DLL. The code in an assembly is not native machine code, but an intermediate language called the Common Intermediate Language (CIL) or IL.



Execution Process

The program's CIL is not compiled to native machine code until it is called to run. At run time, the CLR performs the following steps:

- It checks the assembly's security characteristics.
- It allocates space in memory.
- It sends the assembly's executable code to the Just-in-Time (JIT) compiler, which compiles portions of it to native code.



While applications targeting the .NET Framework interact directly with the FCL, the CLR serves as the underlying engine. In order to understand the .NET Framework, one first must understand the role of the CLR.

Common Language Runtime (CLR)

The CLR is a modern runtime environment that manages the execution of user code, providing services such as JIT compilation, memory management, exception management, debugging and profiling support, and integrated security and permission management.

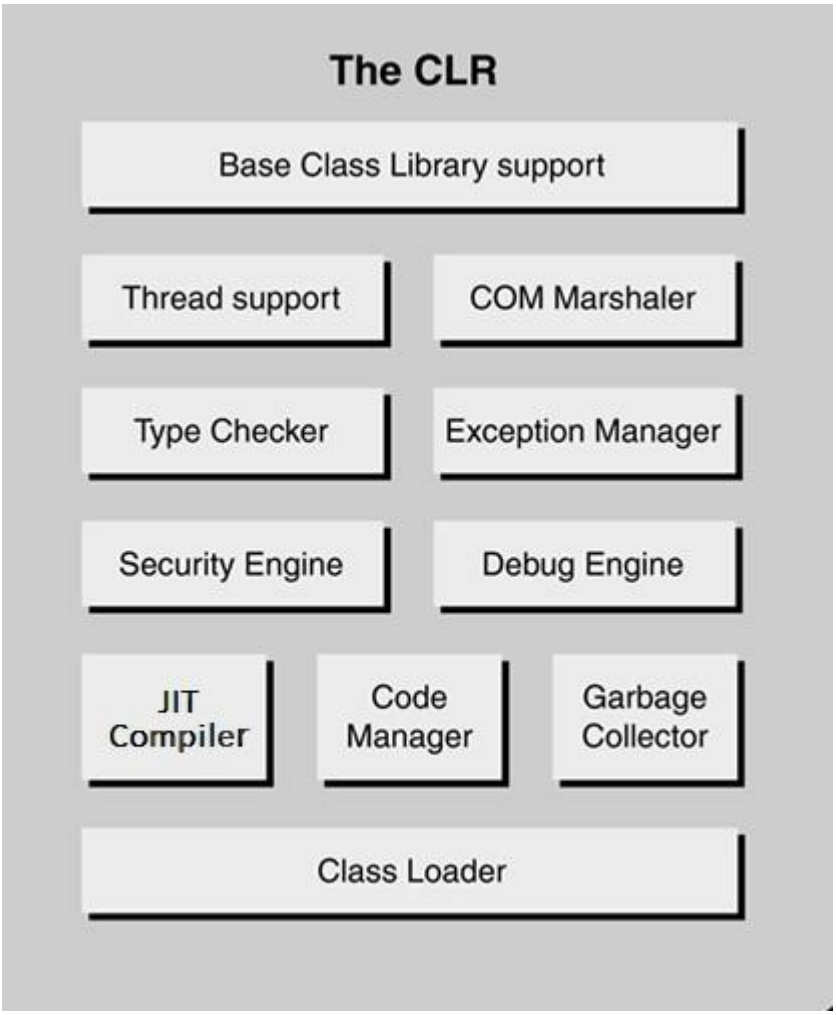
CLR provides an environment to execute .NET applications on target machines. CLR is also a common runtime environment for all .NET code irrespective of their programming language, as the compilers of respective language in .NET Framework convert every source code into a common language known as **MSIL or IL (Intermediate Language)**. CLR also provides various services to execute processes, such as memory management service and security services. CLR performs various tasks to manage the execution process of .NET applications.

The responsibilities of CLR are listed as follows:

- Automatic memory management
- Garbage Collection
- Code Access Security

- Code verification
- JIT compilation of .NET code

Architecture of Common Language Runtime



Class Loader:

Various classes, modules, resources, assemblies etc are loaded by the Class Loader. It loads the modules on demand if they are actually required so that the program initialization time is faster and the resources consumed are lesser.

Garbage Collector:

Automatic memory management is made possible using the garbage collector in CLR. The garbage collector automatically releases objects memory after it is no longer required so that it can be reallocated.

Code Manager:

The code manager in CLR manages the code at run time developed in the .NET framework i.e. the managed code.

JIT Compiler:

The JIT compiler is an important element of CLR, which loads MSIL on target machines for execution. The MSIL is stored in .NET assemblies after the developer has compiled the code written in any .NET-compliant programming language, such as Visual Basic and C#. JIT compiler translates the MSIL code of an assembly and uses the CPU architecture of the target machine to execute a .NET application. It also stores the resulting native code so that it is accessible for subsequent calls. If a code executing on a target machine calls a non-native method, the JIT compiler converts the MSIL of that method into native code. JIT compiler also enforces type-safety in runtime environment of .NET Framework. It checks for the values that are passed to parameters of any method.

Debug Engine:

An application can be debugged during the run-time using the debug engine. There are various ICorDebug interfaces that are used to track the managed code of the application that is being debugged.

Security Engine:

The security engine in the CLR handles the security permissions at various levels such as the code level, folder level, and machine level. This is done using the various tools that are provided in the .NET framework.

Exception Manager:

The exception manager in the CLR handles the exceptions regardless of the .NET Language that created them. For a particular application, the catch block of the exceptions are executed in case they occur and if there is no catch block then the application is terminated.

Type Checker:

Type safety is provided by the type checker by using the Common Type System (CTS) and the Common Language Specification (CLS) that are provided in the CLR to verify the types that are used in an application.

COM Marshaller:

Communication with the COM (Component Object Model) component in the .NET application is provided using the COM marshaller. This provides the COM interoperability support.

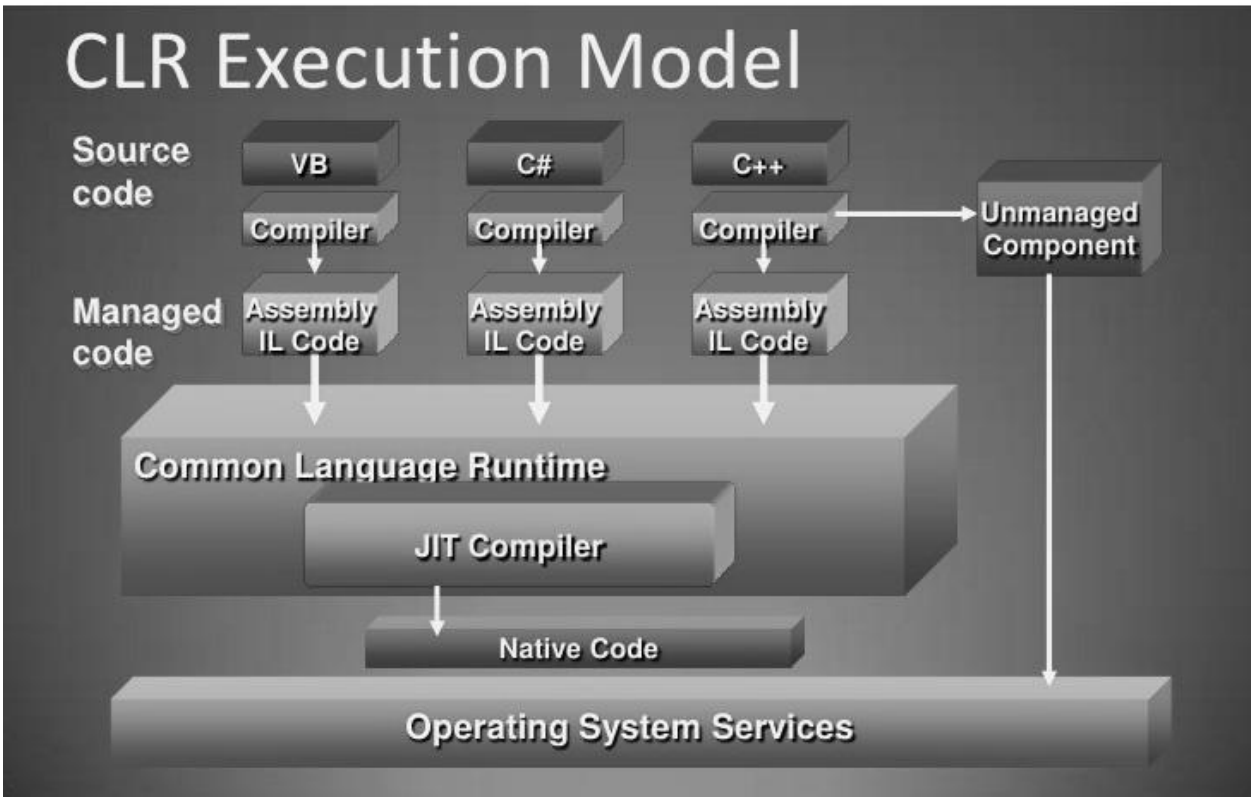
Thread Support:

The CLR provides thread support for managing the parallel execution of multiple threads. The System.Threading class is used as the base class for this.

Base Class Library Support:

The Common Language Runtime provides support for the base class library. The BCL contains multiple libraries that provide various features such as Collections, I/O, XML, DataType definitions, etc. for the multiple .NET programming languages.

Compilation and Execution Model



.NET applications are written in the C#, F#, or Visual Basic programming language. Code is compiled into a language-agnostic Common Intermediate Language (CIL). The code will get compiled with respective compiler (CSC compiler OR VB compiler), based on whether the code is using C# as a language or VB. Compiled code is stored in assemblies—files with a .dll or .exe file extension.

When an app runs, the CLR takes the assembly and uses a just-in-time compiler (JIT) to turn it into machine code that can execute on the specific architecture of the computer it is running on.

Managed and Unmanged Code

Managed code is the code that is executed directly by the CLR instead of the operating system. The code compiler first compiles the managed code to intermediate language (IL) code, also called as MSIL code. This code doesn't depend on machine configurations and can be executed on different machines.

Unmanaged code is the code that is executed directly by the operating system outside the CLR environment. It is directly compiled to native machine code which depends on the machine configuration. In the managed code, since the execution of the code is governed by CLR, the runtime provides different services, such as garbage collection, type checking, exception handling, and security support. These services help provide uniformity in platform and language-independent behavior of managed code applications. In the unmanaged code, the allocation of memory, type safety, and security is required to be taken care of by the developer. If the unmanaged code is not properly handled, it may result in memory leak.

Framework Class Library (FCL)

The Framework Class Library (FCL) is a component of Microsoft's .NET Framework, it is a collection of reusable classes, interfaces and value types, and includes an implementation of the Base Class Library (BCL). The FCL provides a diverse array of higher-level software services, addressing the needs of modern applications. Conceptually, these can be grouped into several categories such as:

Support for core functionality, such as interacting with basic data types and collections;

console, **network** and file I/O; and interacting with other runtime-related facilities.

Support for interacting with **databases**; consuming and producing XML; and manipulating tabular and tree-structured data.

Support for building **web-based** (thin client) applications with a rich server-side event model.

Support for building **desktop-based** (thick client) applications with broad support for the Windows GUI.

Support for building **SOAP-based** XML web services.

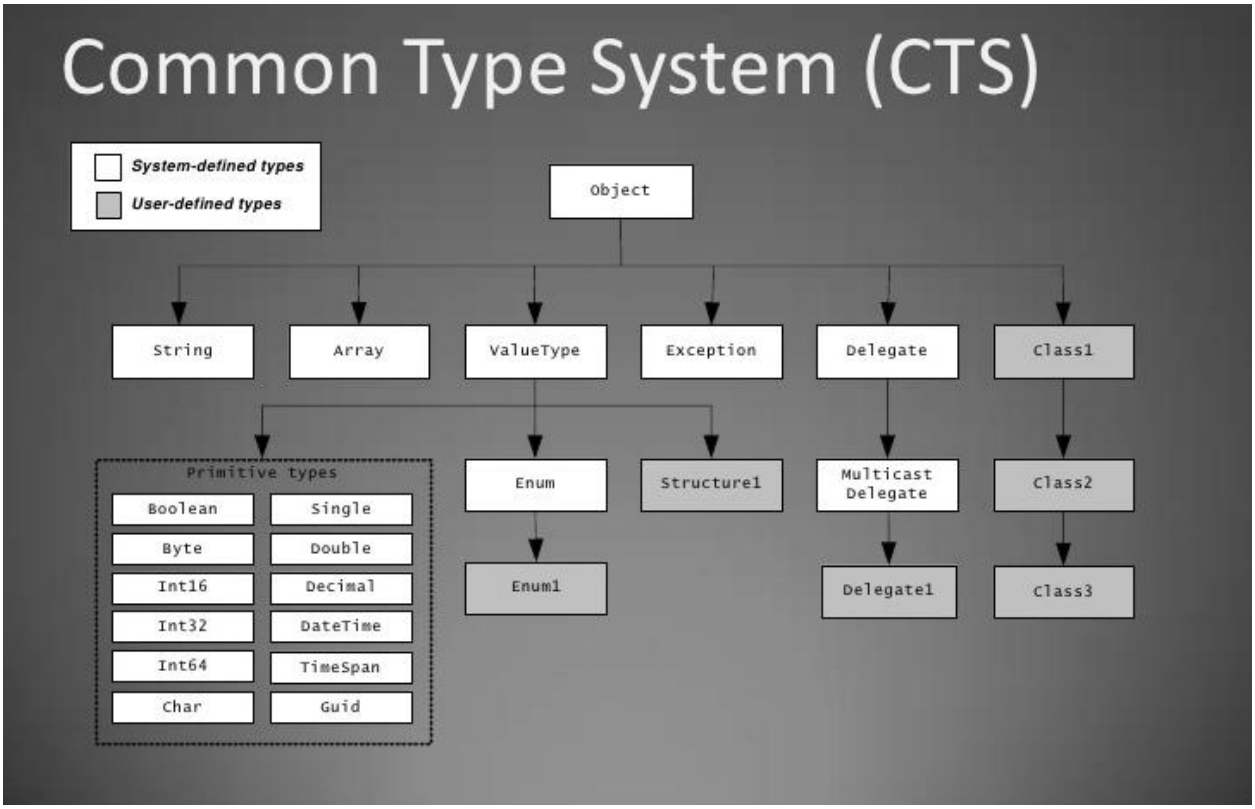
The FCL is vast, including more than 3,500 classes

Common Type System (CTS)

Ultimately, the CLR exists to safely execute managed code, regardless of source language. In order to provide for cross-language integration, to ensure type safety, and to provide managed execution services such as JIT compilation, garbage collection, exception management, etc., the CLR needs intimate knowledge of the managed code that it is executing. To meet this requirement, the CLR defines a shared type system called the Common Type System (CTS). The CTS defines the rules by which all types are declared, defined and managed, regardless of source language. The CTS is designed to be rich and flexible enough to support a wide variety of source languages, and is the basis for cross-language integration, type safety, and managed execution services.

Compilers for managed languages that wish to be first-class citizens in the world of the CLR are responsible for mapping source language constructs onto the CTS analogs. In cases in which there is no direct analog, the language designers may decide to either adapt the source language to better match the CTS (ensuring more seamless cross-language integration), or to provide additional plumbing that preserves the original semantics of the source language (possibly at the expense of cross-language integration capabilities).

Since all types are ultimately represented as CTS types, it now becomes possible to combine types authored in different languages in new and interesting ways. For example, since managed languages ultimately declare CTS types, and the CTS supports inheritance, it follows that the CLR supports cross-language inheritance.



Common Language Specification

Not all languages support the exact same set of constructs, and this can be a barrier to cross-language integration. Consider this example: Language A allows unsigned types (which are supported by the CTS), while Language B does not. How should code written in Language B call a method written in Language A, which takes an unsigned integer as a parameter?

The solution is the Common Language Specification (CLS). The CLS defines the reasonable subset of the CTS that should be sufficient to support cross-language integration, and specifically excludes problem areas such as unsigned integers, operator overloading, and more.

Each managed language decides how much of the CTS to support. Languages that can consume any CLS-compliant type are known as CLS Consumers. Languages which can extend any existing CLS-compliant type are known as CLS Extenders. Naturally, managed languages are free to support CTS features over and above the CLS, and most do. As an example, the C# language is both a CLS Consumer and a CLS Extender, and supports all of the important CTS features.

The combination of the rich and flexible CTS and the widely supported CLS has led to many languages being adapted to target the .NET platform. Microsoft is offering compilers for various

managed languages (C#, VB.NET, JScript, Managed Extensions for C++, Microsoft IL, and J#), and a host of other commercial vendors and academics are offering managed versions of languages, such as COBOL, Eiffel, Haskell and more.

Side-by-Side Execution in the .NET Framework

Side-by-side execution is the ability to install multiple versions of component on the same computer so that an application can choose which version of the common language runtime or of a component it uses.

Basic C# Programming Language Constructs

Introduction of C#

C# is a general purpose, type-safe, object oriented programming language developed by Microsoft . It includes all the important features of a statically typed modern object oriented language and balances simplicity, expressiveness and performance. The C# language is platform neutral.

C# is a rich implementation of the object-orientation paradigm, which includes encapsulation, inheritance, and polymorphism. Encapsulation means creating a boundary around an object to separate its external (public) behavior from its internal (private) implementation details. Following are the distinctive features of C# from an object-oriented perspective:

The fundamental building block in C# is an encapsulated unit of data and functions called a type. C# has a unified type system in which all types ultimately share a common base type. This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic functionality. For example, an instance of any type can be converted to a string by calling its ToString method.

Classes and interfaces

In a traditional object-oriented paradigm, the only kind of type is a class. In C#, there are several other kinds of types, one of which is an interface. An interface is like a class that cannot hold data. This means that it can define only behavior (and not state), which allows for multiple inheritance as well as a separation between specification and implementation.

Properties, methods, and events

In the pure object-oriented paradigm, all functions are methods. In C#, methods are only one kind of function member, which also includes properties and events (there are others, too). Properties are function members that encapsulate a piece of an object's state, such as a button's color or a label's text. Events are function members that simplify acting on object state changes.

Although C# is primarily an object-oriented language, it also borrows from the functional programming paradigm; specifically:

Functions can be treated as values Using delegates, C# allows functions to be passed as values to and from other functions.

Type Safety

C# is primarily a type-safe language, meaning that instances of types can interact only through protocols they define, thereby ensuring each type's internal consistency. For instance, C# prevents you from interacting with a string type as though it were an integer type. More specifically, C# supports static typing, meaning that the language enforces type safety at compile time. This is in addition to type safety being enforced at runtime. Static typing eliminates a large class of errors before a program is even run. It shifts the burden away from runtime unit tests onto the compiler to verify that all the types in a program fit together correctly. This makes large programs much easier to manage, more predictable, and more robust. Furthermore, static typing allows tools such as IntelliSense in Visual Studio to help you write a program, because it knows for a given variable what type it is, and hence what methods you can call on that variable. Such tools can also identify everywhere in your program that a variable, type, or method is used, allowing for reliable refactoring.

C# is also called a strongly typed language because its type rules are strictly enforced (whether statically or at runtime). For instance, you cannot call a function that's designed to accept an integer with a floating-point number, unless you first explicitly convert the floating point number to an integer. This helps prevent mistakes.

Memory Management

C# relies on the runtime to perform automatic memory management. The Common Language Runtime has a garbage collector that executes as part of your program, reclaiming memory for objects that are no longer referenced. This frees programmers from explicitly deallocating the memory for an object, eliminating the problem of incorrect pointers encountered in languages such as C++. C# does not eliminate pointers: it merely makes them unnecessary for most programming tasks. For performance-critical hotspots and interoperability, pointers and explicit memory allocation are permitted in blocks that are marked unsafe.

Platform Support

Historically, C# was used almost entirely for writing code to run on Windows platforms.
However, Microsoft and other companies have since invested in other platforms:

The .NET Core Framework enables web application development in Linux and macOS (as well as Windows).

Xamarin enables mobile app development for iOS and Android.

Blazor compiles C# to web assembly that can run in a browser.And on the Windows platform:

.NET Core 3 enables rich-client and web application development on Windows 7 to 10.

Universal Windows Platform (UWP) supports Windows 10 desktop and devices such as Xbox, Surface Hub, and Hololens.

C# demands that all program logic be contained within a type definition (a type is a general term referring to a member of the set {class, interface, structure, enumeration, delegate}). Unlike many other languages, in C# it is not possible to create global functions or global points of data. Rather, all data members and all methods must be contained within a type definition.

A Sample C# Program

```
using System;    // Importing namespace
namespace SampleProgram   // Declare a scope
{
     class Program            //  Class declaration
     {
               static void Main(string[] args) // Main Method
               {
                    Console.WriteLine("Sample Output"); // Statement
               }
     }
}
```

// *Namespaces are used in C# to organize and provide a level of separation of codes.*

Output :

Sample Output

C# is a case-sensitive programming language. Therefore, **Main** is not the same as **main**, and **Readline** is not the same as **ReadLine**. Be aware that all C# keywords are lowercase (e.g., public, lock, class, dynamic), while namespaces, types, and member names begin (by convention) with an initial capital letter and have capitalized the first letter of any embedded words (e.g., Console.WriteLine, System.Windows. MessageBox, System.Data.SqlClient).

Every executable C# application (console program, Windows desktop program, or Windows service) must contain a class defining a Main() method, which is used to signify the entry point of the application.

DotNet Tool

The dotnet tool (dotnet.exe on Windows) helps you to manage .NETsource code and binaries from the command line. You can use it to both build and run your program, as an alternative to using an Integrated Development Environment (IDE) such as Visual Studio or Visual Studio Code.

The following command scaffolds a new console project with its basic structure:

> dotnet new Console -n SampleProgram

This creates a folder called SampleProgram containing a project file called SampleProgram.csproj and a C# file called Program.cs with a Main method that prints "Sample Output".

To compile an application, the dotnet tool requires a project file as well as one or more C# files.

To build and run the program from the SampleProgram folder:

> dotnet run SampleProgram OR,

> dotnet run OR,

> dotnet run SampleProgram.csproj

To build the program with out running.

> dotnet build OR,

> dotnet build SampleProgram.csproj

The output assembly will be written to a subdirectory under bin\Debug.

Identifiers and Keywords

Identifiers are names used to identify variables, functions, structures, classes, or any other user-defined items or entities. Identifiers must be unique and different from keywords.They are created to give unique name to identify it during the execution of the program.

Identifiers in the Sample C# Program above.

System, SampleProgram, Program, Main, Console, WriteLine

An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore. C# identifiers are case sensitive. By convention, parameters, local variables, and private fields should be in camel case (e.g., firstName, lastName, _age), and all other identifiers should be in Pascal case (e.g., DisplaySalary, DisplayAge).

Keywords are predefined, reserved words in programming that have special meaning for the compiler. They can not be used as a variable , function or any other user-defined items name.

Keywords in the Sample C# Program above.

using, namespace, class, static, void, string

All these keywords are in lowercase. Here is a complete list of all C# keywords.

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

Comments

A comment is an annotation in a source code with the intention to provide a readable explanation of the code. C# offers two different styles of source-code documentation: singleline comments and multiline comments.

A single-line comment begins with a double forward slash and continues until the end of the line;

For example:

```
int a = 5; // SingleLine comment
```

A multiline comment begins with /* and ends with */;

For example:

```
int a = 5;
/* This is a comment that
spans two lines */
int b = 7;
```

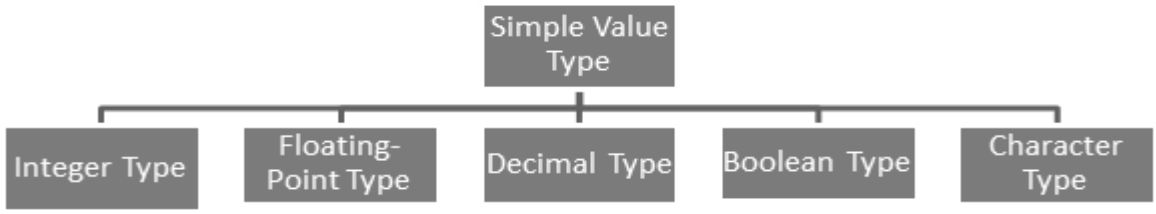
Data Types and Variables

Data Types are an integral part in any programming language. They represent how to express numbers, characters, strings, decimal point values, and so forth.

Like any programming language, C# defines the data type to represent a variable. CLR provide two categories of data type Value Type and Reference Type. Both types are stored in different places of memory; a value type stores its value directly in stack and reference types store a reference to the value in a managed heap.

Value Type:

Value based data types(Value Type) include all numerical data types such as Integer, Float, Byte, and so forth.



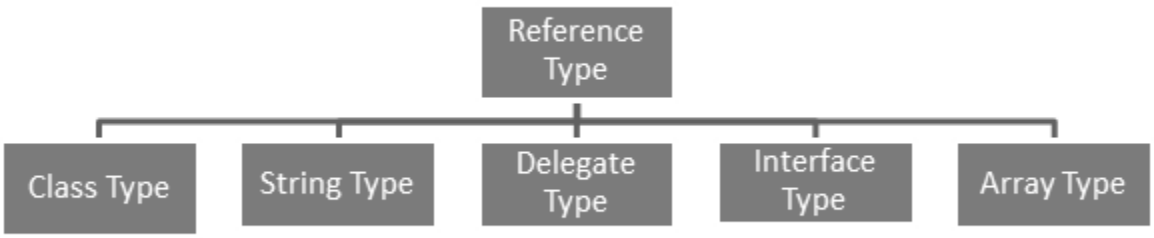
Sample Code:

```
using System;
namespace Basic
{
    class Program
    {
        static void Method(int x)
        {
                x =  200;
                Console.WriteLine(x);
        }
        static void Main(string[] args)
        {
                int x = 100;
                Console.WriteLine(x);
                Method(x);
                Console.WriteLine(x);
        }
    }
}
```

```
Output:
     100
     200
     100
```

Reference Type:

Reference based data types(Reference Type) include Classes, Interfaces, Delegates etc. If the value of one variable gets changed, the value of the other variable also reflects the same value.



Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

```
TestClass   obj1,   obj2;
obj1   =    new TestClass();
obj2  =   obj1;
```

The important point to understand here is that TestClass revolves around objects only. obj1 and obj2 are variables of reference type and both point to the same location that contains this object.

Boxing and Unboxing

The conversion of a value type into a reference type is termed as boxing; the conversion of a reference type into a value type is termed as unboxing.

Variables

A variable denotes a storage location that can contain different values over time. It is a special container that holds some data at specific memory location. It could be numeric, string, date, character, Boolean, etc. C# is a type-safe language, which means that all the variables and constants in C# must have a type. Based on the data type, specific operations can be carried out on the variable.

Variable can be declared by the following syntax in C#:

DataType Identifier;

AccessModifier DataType VariableName;

AccessModifier DataType VariableName = InitialValue;

Sample Code:

```
int x;
public int y;
private int z = 10;
```

Variable can either be initialized in the same statement as the declaration or later in the code.

Sample Code:

```
string  schoolName = “kbc”;
private  DateTime  _currentDateTime;
double  interestRate  =  0.2;
```

Var Keyword

The var keyword specifies implicit typing. The code does not specify the type, and the compiler can infer the type from the initialization. The compiler recognizes that the literal inside the double quotes is a string and declares the variable as such. Implicitly typed variables should not be confused with variant types. var maintains C#'s type-safety mechanisms and does not allow a type change after declaration.

Sample Code:

```
var schoolName  =  “kbc”
var  itemCount  =  10;
```

Literals

A literal is a notation that represents a fixed value. C# literals can be of any of the value types. The way each literal is represented depends upon its type.

Integer Literals

```
int x   =   100;
```

Floating-point Literals

```
double radius  = 3.5
```

Character Literals

```
char ch = 'a';
```

String Literals

String constants, also known as string literals, are a special type of constants which store fixed sequences of characters. A string literal is a sequence of any number of characters surrounded by double quotes:

```
string  color =  @“Red \n Color”;
const string str = "String literal \"1\"";
```

Boolean Literals

```
bool  found  =  true;
```

NULL Literals

```
class Point {...};
Point p  =  null;
```

Type Conversions (Type Casting)

Type Conversion or Type Casting is a mechanism to convert one data type value to another one.C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either implicit or explicit: implicit conversions happen automatically, and explicit conversions require a cast.

Integral type conversions are implicit when the destination type can represent every possible value of the source type. Otherwise, an explicit conversion is required.

Implicit Conversion (automatically)

Implicit conversions are allowed when both of the following are true:

- The compiler can guarantee that they will always succeed.
- No information is lost in conversion.

Sample Code:

```
int  a = 25;    // a is 32 bit integer
long b = a;
// b is 64 bit integer, a is converted to 64-bit implicitly
```

Explicit Conversions

Explicit conversions are required when one of the following is true:

- The compiler cannot guarantee that they will always succeed.
- Information might be lost during conversion.

Sample Code:

```
int a = 25;
short b  = (short) a;
```

Constants

A constant is a field whose value is evaluated statically at compile time and can never be changed. The constant can be any of the built-in numeric types, bool, char, string, or an enum type. User-defined types, including classes, structs, and arrays, cannot be constants.

A constant is declared with const keyword and must be initialized with a value.

Sample Code:

```
const double Pi  =  3.14159;
public const int X = 45, Y = 90, Z = 180;
```

Characters

Char represents a character value type and holds a single Unicode character value. It is 2 bytes in size. Char is a value type since it actually stores the value in the memory that has been allocated on the stack. A char literal is specified within single quotes:

char ch = 'a';

char nLine = '\n';

There are several methods that can be used on the char data type within the .Net framework. These include but not limited to:

char.IsDigit

char.IsLower

char.ToLower

String

A string is a immutable(unmodifiable) sequence of unicode characters that is used to represent text. It can be a character, a word or a long passage surrounded with the double quotes.

```
string ch  =  "a";
string word  =  "word";
string text  =  "This is a string.";
```

A string is immutable in C#. It means it is read-only and can not be changed once created in memory. String is a reference type however, follow value-type semantics.

In C#, a string is a collection or an array of characters. So, string can be created using a char array or accessed like a char array.

```
char[]  vowels  = {'a',  'e',  'i',  'o',  'u'};
foreach (char c in vowels)
{
     Console.WriteLine(c);
}
```

Escape Characters

C# string literals may contain various special characters, C# includes escaping character '\' before these special characters to include in a string.

string a = "Here's a tab:\t";

\t Inserts a horizontal tab

String Literal Escape Characters

\' Inserts a single quotes into a string.

\" Insert double quotes into a string.

\\ Insert a backslash into a string.

\n Insert a new line

\r Insert a carriage return

Verbatim Strings

A verbatim string literal is prefixed with @ and does not support escape sequences.Using verbatim strings, you disable the processing of a literal's escape characters and print out a string as is.

```
String with escape character

string a1 = "\\\\server\\folder\\file.cs";

Verbatim string

string a2 = @"\\server\folder\file.cs";
```

A verbatim string literal can also span multiple lines:

```
string str = @"this is a
multi line
string";
```

Basic String Operations

The string keyword is an alias for String. Therefore, String and string are equivalent, and can be used interchangeably. The String class provides many methods for safely creating, manipulating, and comparing strings.

Working with the member of System.String

```
string programName = “SampleProgram”;
string len = programName.Length;
string  programNameUpper = programName.ToUpper();
string programNameLower = programName.ToLower();
```

String Concatenation

string variables or two strings can be connected together to build larger strings using “+” operator.

```
string str1  =  “C#”;
string str2   =  “Programming”;
string  str3  =   str1  +   str2;
```

It is possible to perform string concatenation by calling String.Concat()

```
String str3  =  String.Concat(str1, str2);
```

The operands might be a nonstring value, in which case ToString() is called on that value:

```
string str  =  "s"  +  5;  // output  =>  s5
```

String Interpolation

C# string interpolation is a method of concatenating, formatting and manipulating strings. A string preceded with the $ character is called an interpolated string.

```
int x  =  4;
Console.Write ($"A square has {x} sides");
```

Interpolated strings must complete on a single line, unless you also specify the verbatim string operator:

```
int x  =  4;
string s = $@"A square has {
x} sides";
```

Strings Are Immutable

One of the interesting aspects of System.String is that after you assign a string object with its initial value, the character data cannot be changed. At first glance, this might seem like a flat-out lie, given that you are always reassigning strings to new values and because the System.String type defines a number of methods that appear to modify the character data in one way or another (such as uppercasing and lowercasing). However, if you look more closely at what is happening behind the scenes, you will notice the methods of the string type are, in fact, returning you a new string object in a modified format.

```
static void StringsAreImmutable()
{
      // Set initial string value.
      string s1 = "This is my string.";
      Console.WriteLine("s1 = {0}", s1);
      // Uppercase s1?
      string upperString = s1.ToUpper();
      Console.WriteLine("upperString = {0}", upperString);
      // Nope! s1 is in the same format!
      Console.WriteLine("s1 = {0}", s1);
}
```

If you examine the relevant output that follows, you can verify that the original string object (s1) is not uppercased when calling ToUpper(). Rather, you are returned a copy of the string in a modified format.

s1 = This is my string.

upperString = THIS IS MY STRING.

s1 = This is my string.

The same law of immutability holds true when you use the C# assignment operator. To illustrate, implement the following StringsAreImmutable2() method:

```
static void StringsAreImmutable2()
{
      string s2 = "My other string";
      s2 = "New string value";
}
```

C# Iteration Constructs

Iteration constructs(Loops) in C# essentially allow you to execute a block of code repeatedly until a certain condition is met. C# provides the following four iteration constructs:

- **for loop**
- **foreach/in loop**
- **while loop**
- **do/while loop**

The for Loop

The for loop is to iterate over a block of code a fixed number of times where you are able to specify how many times a block of code repeats itself, as well as the terminating condition. The for loop construct syntax is as shown below:

```
for (initialization;   condition;   increment / decrement)
{
      //Statements
}
```

The foreach / in Loop

The foreach loop is to iterate through each item in to a collection without the need to test for an upper limit. The foreach loop will iterate the collection only in a linear (n + 1) fashion.

The syntax for a foreach construct is as shown below:

```
foreach (Type Identifier in expression)
{
   // Statements
}
```

Sample Code:

```
string[] brands  =  {"HP", "Dell", "Acer", "Lenevo" };
foreach (string b in brands )
{
      Console.WriteLine(b);
}
```

The while Loop

The while loop repeatedly executes a body of code till the specified condition is true. The expression is tested before the body of the loop is executed:

The syntax for a C# while loop is as shown below:

```
while (condition)
{
   // Statements
}
```

Sample Code:

```
int i = 1;
while (i < 5)
{
      Console.WriteLine (i);
      i++;
}
```

The do while

The do while loop is very similar to a while loop except that do/while loop is guaranteed to execute the corresponding block of code at least once.

The syntax for a C# do loop is as shown below:

```
do
{
   // Statements
} while (condition)
```

Sample Code:

```
int i = 1;
do
{
      Console.WriteLine (i);
      i++;
} while ( i  < 5 );
```

Decision Constructs (Conditional Statement)

C# defines two simple constructs to alter the flow of the program, based on various circumstances. C# has two constructs for branching code, the if statement and the switch statement.

The if/else statement

The if statement contains a boolean condition followed by a single or multi-line code block to be executed. At runtime, if a boolean condition evaluates to true, then the code block will be executed, otherwise else block will be executed. The syntax for the if/else construct is as shown below:

```
if (expression)
{
   // Statements if expression evaluates to True
}
else
{
   // Statements if expression evaluates to False
}
```

The Conditional Operator

The conditional(ternary) operator (?:) is a shorthand method of writing a simple if-else statement. The syntax is as shown below:

condition ? first_expression : second_expression;

Sample Code:

```
int x = 100, y = 10;
var result  =  x > y ? "x is greater than y" : "x is less than
y";
```

```
Console.WriteLine(result);
```

The switch Statement

The switch statement is very similar to the one we used in C, except that in C# it is mandatory to specify a break statement for each and every case block, without which the compiler will generate an error. the switch statement allows you to handle program flow based on a predefined set of choices. The syntax is as shown below:

```
switch (variable)
{
   case value1:
      // Statements
     break;
   case value2:
      // Statements
     break;
   default:
      // Statements
     break;
}
```

Sample Code:

```
Console.WriteLine("C# or JAVA");
Console.Write("Please pick your language preference: ");
string langChoice = Console.ReadLine();
switch (langChoice)
{
     case  "C#":
           Console.WriteLine("Good choice, C# is a fine
           language.");
           break;
     case  "JAVA":
           Console.WriteLine("JAVA: OOP, multithreading and
           more!");
           break;
     default:
           Console.WriteLine("Something else!");
           break;
}
```

C# 8 supports a new interesting feature called **switch expressions** where the cases are expressions.

Syntax improvements for the switch expressions:

- Instead of switch (variable), the variable comes before the switch keyword.
- The patterns are now comma-separated list (,).
- The lambda arrow => replaces the case keyword and the colon (:). It is used between the pattern and the expression.
- The left side of the => is a pattern with an optional when clause to add additional conditions. The right side of the => is an expression without a return keyword.
- The switch expressions do not support break and return keywords.
- The underscore (_) character replaces the default keyword to signify that it should match anything if reached.
- The bodies are now expressions instead of statements. The selected body becomes the switch expression's result.
- Only a single expression can be used on the right side of the arrow and the expression value needs to be returned as a result.

Above code for switch block can be re-written with switch expression as shown below:

```
Console.WriteLine("C# or JAVA");
Console.Write("Please pick your language preference: ");
string langChoice = Console.ReadLine();
var result = langChoice switch
{
      "C#" => "Good choice, C# is a fine language.",
      "JAVA" => " JAVA: OOP, multithreading and more!",
       _   => " Something else !"
};
Console.WriteLine(result);
```

C# 8 supports three new patterns TUPLE, POSITIONAL, AND PROPERTY PATTERNS, mostly for the benefit of switch statements/expressions

Tuple patterns let us switch on multiple values as shown below:

```
int cardNumber = 12; string suite = "spades";
string cardName = (cardNumber, suite) switch
{
      (13, "spades") => "King of spades",
      (13, "clubs") => "King of clubs",
      //more code
};
```

Positional patterns allow a similar syntax for objects that expose a deconstructor, and **property** patterns let us match on an object's properties.

Understanding C# Arrays

An array represents a fixed number of variables (called elements) of a particular type. The elements in an array are always stored in a consecutive block of memory, providing highly efficient access.

An array is denoted with square brackets after the element type:

```
string[ ] items;   //String array
// Declaration of an integer array with size 5
int[ ] oddNums  =  new int[5];
```

An array initialization expression lets you declare and populate an array in a single step:

```
char[ ] vowels  =  new char[ ] {'a', 'e', 'i', 'o', 'u'};
or simply:
char[ ] vowels  =  {'a', 'e', 'i', 'o', 'u'};
int[ ] evenNums  =  new int[5] { 2,  4,  6,  8,  10 };
```

Accessing Array Elements

Array elements can be accessed with array indexes, placed index of the element within square brackets with the array name.

```
oddNums[0] = 1;
oddNums[1] = 3;
oddNums[2] = 5;
```

```
oddNums[3] = 7;
oddNums[4] = 9;
Console.WriteLine(oddNums[3]);
```

Accessing an array elements using loops

The for loop can be used to access array elements, and use the Length property of an array in conditional expression of the for loop.

Sample Code:

```
int[ ] evenNums = { 2,  4,  6,  8,  10 };
for (int i  =  0;  i < evenNums.Length;   i++)
{
     Console.WriteLine(evenNums[ i ]);
}
```

The Length property of an array returns the number of elements in the array.

Use foreach loop to access elements an array without using index.

Sample Code:

```
char[ ] vowels  =  {'a', 'e', 'i', 'o', 'u'};
foreach (var v  in vowels)
{
      Console.WriteLine(v);
}
```

Multidimensional Arrays

Multidimensional arrays come in two varieties: rectangular and jagged. The multidimensional array can be declared by adding commas in the square brackets.

Rectangular Arrays

Rectangular arrays are declared using commas to separate each dimension. The following declares a rectangular two-dimensional array for which the dimensions are 2 by 2:

```
int[ , ]   intArray =  new int[2, 2]; //OR
int[ , ]   intArray = new int[2,2] {
                                      {1,  2},
                                      {3,  4}
                                    };
intArray[0, 0]; //returns 1
intArray[0, 1]; //returns 2
```

```
intArray[1, 0]; //returns 3
intArray[1, 1]; //returns 4
```

Jagged Arrays

A jagged array can be defined as an array consisting of arrays. The jagged arrays are used to store arrays instead of other data types. A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array, and the second bracket specifies the dimensions of the array which is going to be stored.

```
int[ ][ ] matrix = new int[2][ ]; // OR
int[ ][ ]  matrix  = new int[2][ ] {
                new int[3]{1,  2,  3},
                new int[4]{4,  5,  6,  7}
            };
```

The inner dimensions aren't specified in the declaration because, unlike a rectangular array, each inner array can be an arbitrary length.

A jagged array can be accessed using two for loops, as show below.

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

Indices And Ranges

The caret (^) and range operator (..) provide a different syntax for accessing elements in an array: Span, or ReadOnlySpan. The range operator is used to specify the start and end of a range for a sequence.

Indices let us refer to elements relative to the end of an array by using the ^ operator. ^1 refers to the last element, ^2 refers to the second-to-last element, and so on:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char lastElement = vowels [^1]; // 'u'
char secondToLast = vowels [^2]; // 'o'
```

C# implements indices with the help of the Index type, so you can also do the following:

```
Index first = 0;
Index last = ^1;
char firstElement = vowels [first]; // 'a'
char lastElement = vowels [last]; // 'u'
```

Ranges let us "slice" an array by using the .. operator:

```
char[] firstTwo = vowels [..2]; // 'a', 'e'
char[] lastThree = vowels [2..]; // 'i', 'o', 'u'
char[] middleOne = vowels [2..3] // 'i'
char[] lastTwo = vowels [^2..]; // 'o', 'u'
```

C# implements indexes and ranges with the help of the Index and Range types:

```
Index last = ^1;
Range firstTwoRange = 0..2;
char[] firstTwo = vowels [firstTwoRange]; // 'a', 'e'
```

Sample Code:

```
string[] techArray = { "C", "C++", "C#", "F#", "JavaScript",
"Angular", "TypeScript", "React", "GraphQL" };

 Range range = 1..4;

 foreach (var item in techArray[range])
 {
     Console.WriteLine(item);  //C++ C# F#
 }
```

Classes

A class is a user-defined type that is composed of field data (often called member variables) and members that operate on this data (such as constructors, properties, methods, events, and so forth).

In C#, a class can be defined by using the class keyword.

```
class  ClassName
{

}
```

Fields

A field is a variable that is a member of a class or struct; for example:

```
class Student
{
      string name;
      public int age = 10; //Initialization
}
```

Field initialization is optional. An uninitialized field has a default value (0, \0, null, false). Field initializers run before constructors.

Properties

A property in C# is a member of a class that provides a flexible mechanism for classes to expose private fields. C# Properites doesn't have storage location. C# Properites are extension of fields and accessed like fields.

Sample Code:

```
class Point
{
    private int x;
    public int X
    {
           get
           {
                return x;
           }
           set
           {
                x  =  value;
           }
    }
}
```

C# also provides a way to use short-hand properties, where you do not have to define the field for the property, and you only have to write get; and set; inside the property.

```
class Point
{
     public int X { get;  set; }
}
```

Indexers

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers are defined with **this** keyword and similar to properties but are accessed via an index argument rather than a property name.

For example the string class has an indexer that lets you access each of its char values via an int index:

```
string s = "hello";
Console.WriteLine (s[0]); // 'h'
Console.WriteLine (s[3]); // 'l'
```

Sample Code:

```
class Employee
{
    private string[] arr = new string[2];
    public string this[int i]
    {
        get
        {
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}
public class Program
{
    public static void Main(string[] args)
    {
```

```
            Employee eid = new Employee();
            eid[0] = "EMP-001";
            eid[1] = "EMP-002";
            for (int i = 0; i < 2; i++)
            {
                Console.WriteLine(eid[i]);
            }
        }
    }
```

The syntax for using indexers is like that for using arrays, except that the index argument(s) can be of any type(s). Indexers must have at least one parameter else a compile-time exception will be generated. Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not necessary that the indexes have to be integers. C# allows indexes to be of other types.

Indexers Overview

- Indexers enable objects to be indexed in a similar manner to arrays.
- A get accessor returns a value. A set accessor assigns a value.
- The this keyword is used to define the indexer.
- The value keyword is used to define the value being assigned by the set accessor.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Constructors

A constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created. C# supports the use of constructors, which allow the state of an object to be established at the time of creation. A constructor is a special method of a class that is called indirectly when creating an object using the new keyword. However, unlike a “normal” method, constructors never have a return value (not even void) and are always named identically to the class they are constructing.

```
class ClassName
{
    public ClassName()
    {
    }
}
```

A constructor without any parameters is called a default constructor. It is invoked at the time of creating object. The default constructor initializes all numeric fields to zero and all string and object fields to null inside a class.

The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class with different values. The default constructor initializes:

```
class Student
{
      string name;
      string collegeName;
      public Student()
      {
            collegeName = “kbc”;
      }
}
```

A constructor with at least one parameter is called as parameterized constructor. The advantage of a parameterized constructor is that you can initialize each instance of the class with a different value.

Sample Code:

```
class Student
{
      string name;
      string collegeName;
      public Student(string studentName)
      {
            collegeName = “KBC”;
            name = studentName;
      }
}
```

A constructor that is preceded by a private access modifier is called a private constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class

```
class Student
{
      string name;
      private Student()
      {
      }
      Public Student(string studentName)
      {
            name = studentName;
      }
}
```

Nested Classes

A nested class is a class declared with in another class. It is a member of its enclosing class and the members of an enclosing class have no access to members of a nested class.

```
class  A {     // Enclosing Class
   public string name;
   public class B {  // Nested Class
      public int count;
   }
}
```

Accessing nested class

```
A.B  b = new A.B();
b.count = 1;
```

Access Modifiers

Access modifiers are the keywords that are used to specify accessibility or scope of variables and functions in the C# Program. To promote encapsulation, a type or type member can limit its accessibility to other types and other assemblies by adding one of six access modifiers to the declaration:

private

Member is accessible inside the type only. This is the default.

internal

Member is accessible inside the type and any type in the same assembly.

protected

Member is accessible inside the type and any type that inherits from the type.

public

Member is accessible everywhere.

internal protected

Member is accessible inside the type, any type in the same assembly, and any type that inherits from the type. Equivalent to a fictional access modifier named internal_or_protected.

private protected

Member is accessible inside the type, or any type that inherits from the type and is in the same assembly. Equivalent to a fictional access modifier named internal_and_protected. This combination is only available with C# 7.2 or later.

Methods

Methods normally belongs to a class, and they define how an object of a class behaves. A method performs an action in a series of statements. A method can receive input data from the caller by specifying parameters and output data back to the caller by specifying a return type. A method can specify a void return type, indicating that it doesn't return any value to its caller. A method can also output data back to the caller via ref/out parameters.

Sample Code:

```
int  Addition(int a, int b)
{
     return a + b;
}
```

A type can overload methods (have multiple methods with the same name) as long as the signatures are different.

```
void Print(int x) { }
void Print(double x) {  }
void Print(int x, float y) { }
void Print(float x, int y) {  }
```

Parameters

A method may have a sequence of parameters. Parameters define the set of arguments that must be provided for that method.

Sample Code:

```
static int Add (int a, int  b, int c)
{
     return a + b + c;
}
```

In the example above, the method Add has mulitple parameters name a, b, c of type int.

Method parameters have modifiers available to change the desired outcome of how the parameter is treated. Each method has a specific use case:

ref is used to state that the parameter passed may be modified by the method.

in is used to state that the parameter passed cannot be modified by the method.

out is used to state that the parameter passed must be modified by the method.

```
static void Increment(ref  int  count)
{
      count  =  count + 1;
}
```

Inheritance

The inheritance is an aspect of OOP that facilitates code reuse. It is the mechanism in C# by which one class is allowed to inherit the features (fields and methods) of another class. One of the classes is called a child class, and the other is called a parent class. The child class inherits from the parent class. Conversely, we can also call the classes derived class and base class, respectively.

```
public class Parent
{
      public void Print()
      {
                  Console.WriteLine("Base class");
      }
}

public class Child : Parent
{
}
```

In inheritance, the derived class inherits all the members (fields, methods) of the base class, but derived class cannot inherit the constructor of the base class.

C# demands that a given class have exactly one direct base class. It is not possible to create a class type that directly derives from two or more base classes.

```
public class Car  :  Honda,  Toyota         // C# does not allow.
{
}
```

In inheritance, the base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Sample Code:

```
public class Vehicle
 {
     public Vehicle()
     {
         Console.WriteLine("Vehicle");
```

```
    }
}
public class Car : Vehicle
{
    public Car()
    {
        Console.WriteLine("Car");
    }

}

class Program
{
    static void Main(string[] args)
    {
        Car d = new Car();
    }
}
```

Use of base keyword

The base keyword is used to access members and functionalities of the base class from within a derived class: It serves two essential purposes.

- Accessing an overridden function member from the subclass.
- Calling a base-class constructor.

Sample Code:

```
public class Animal
{
      public virtual void Eat()
      {
            Console.WriteLine(“Animal: Eat()");
      }
}

public class Cat  :  Animal
{
      public override void Eat()
      {
                  base.Eat();
                  Console.WriteLine("Cat: Eat()");
```

```
        }
    }
```

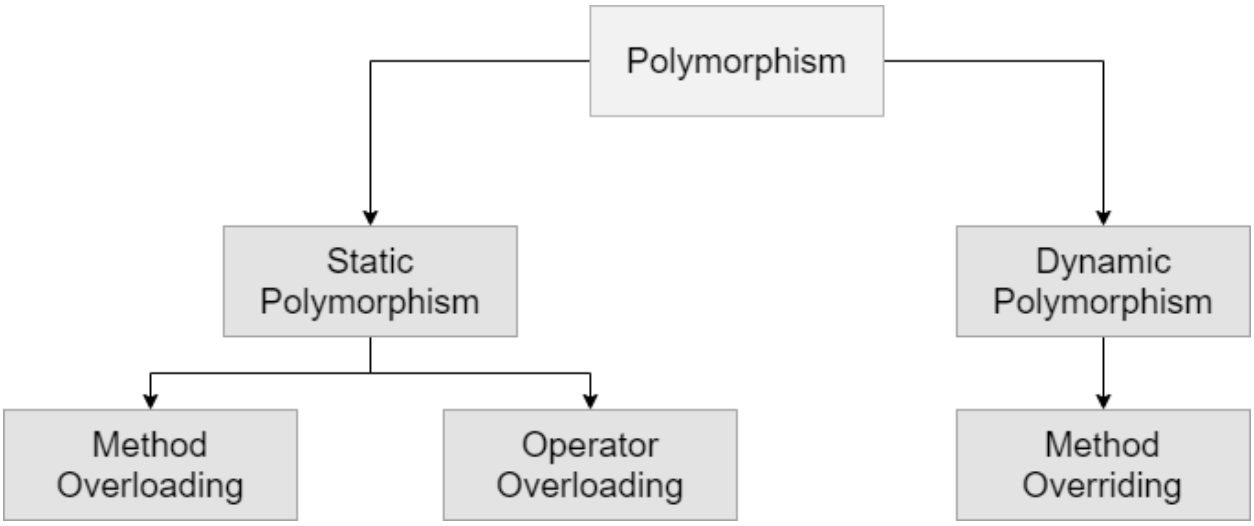
If a base class and a derived class define same method, **base** keyword is used to access the method of the base class. If derived class doesn't define the same method, there is no need to use base keyword.

Polymorphism

Polymorphism is one of the core principles of Object Oriented Programming, which is the ability for classes to provide different implementations of methods that are called by the same name. Polymorphism allows a method of a class to be called without regard to what specific implementation it provides.

There are two types of polymorphism in C#:

- Static / Compile Time Polymorphism
- Dynamic / Runtime Polymorphism



Compile Time polymorphism (Static)

Compile time polymorphism is also known as early binding and static polymorphism. Method overloading and operator overloading are two techniques of implementing compile time

polymorphism. Method overloading means creating two or more methods on the same type that differ only in the number or type of parameters but have the same name. Such methods perform a different task on the various input parameters. Do not use ref or out modifiers to overload members.

Compile time polymorphism is known as early binding because the compiler has the knowledge of different overloaded method names at compile time. The compiler also knows beforehand which method must be executed when called.

```
   public class Interest
    {
        public double Calculate(double amount, double rate)
        {
            return amount + (amount * rate);
        }
        public double Calculate(double amount, double rate, string
holdertype)
        {
            return amount + (amount * rate) + 2000;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Interest i = new Interest();

            double finalamount = i.Calculate(5000.00, 0.1);
                  Console.WriteLine("Normal interest for a holder " +
     finalamount);

            finalamount = i.Calculate(5000.00, 0.1, "prime");
            Console.WriteLine("Prime interest for a holder " +
finalamount);
        }
    }
```

Runtime polymorphism (Dynamic)

Runtime polymorphism is also known as late binding and dynamic polymorphism. Abstract classes and virtual functions (method overriding) are the two techniques of implementing runtime polymorphism. Derived class can override a base class method to suit its needs. The

methods of the base class that are supposed to be overridden are declared as virtual methods. The virtual methods, therefore, can be implemented in different ways in the derived classes. The call to these functions is decided at runtime.

```
class Base {
      public virtual string methodOverride(){
            return “base method to overide”;
      }
      public string method(){
            return “base simple method “;
      }
      public string methodToHide(){
            return “base method to hide”;
      }
      public virtual string methodToHideWithVirtual(){
            return “base method to hide With Virtual”;
      }
}


class Derived : Base {
      public override string methodOverride() {
            return “derived class: method overidded”;
      }
      public string method() {
            return “derived class: simple method”;
      }
      public new string methodToHide() {
            return “derived class: method hidden”;
      }
      public new string methodToHideWithVirtual() {
            return “derived class: method with virtual hidden”;
      }
}
```

Sample Code:

```
public class Interest
    {
        public virtual double Calculate(double amount, double
rate)
        {
```

```
            return amount + (amount * rate);
        }
    }
    public class SimpleInterest : Interest
    {
        public override double Calculate(double amount, double
rate)
        {
            return amount + (amount * rate) + 1000;
        }
    }
    public class FixedInterest : Interest
    {
        public override double Calculate(double amount, double
rate)
        {
            return amount + (amount * rate) + 1500;
        }
    }
    public class Program
    {
        public static void Main(string[] args)
        {
            Interest i = new Interest();
            double finalamount = i.Calculate(5000.00, 0.1);
            Console.WriteLine("Normal interest for a holder " +
finalamount);

            i = new SimpleInterest();
            finalamount = i.Calculate(5000.00, 0.1);
            Console.WriteLine("Simple interest for a holder " +
finalamount);

            i = new FixedInterest();
            finalamount = i.Calculate(5000.00, 0.1);
            Console.WriteLine("Fixed interest for a holder " +
finalamount);
        }
   }
```

Method Overloading

Overloading allows methods to have the same name but with different parameters. Method overloading falls under the static poloymorphism.The process of creating more than one method in a class with same name or creating a method in derived class with same name as a method in base class is called as method overloading. No need to use any keyword while overloading a method either in same class or in derived class.

- Overloaded methods are differentiated based on the number and type of the parameters passed as arguments to the methods.
- When defining overloaded methods, a rule must be followed i.e, the overloaded methods must differ either in number of parameters they take or the data type of at least on parameter.
- The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error. If both methods have the same parameter types, but different return type, then it is not possible.

Sample Code:

```
class Calculate
{
    public void AddNumbers(int a, int b)
    {
        Console.WriteLine("a + b = {0}", a + b);
    }
    public void AddNumbers(int a, int b, int c)
    {
        Console.WriteLine("a + b + c = {0}", a + b + c);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Calculate c = new Calculate();
        c.AddNumbers(1, 2);
        c.AddNumbers(1, 2, 3);
    }
}
```

The virtual and override Keywords

Polymorphism provides a way for a derived class to define its own version of a method defined by its base class, using the **override** keyword in C#. The process of re-implementing the base class non-static method in the derived class with the same prototype (same signature defined in the base class) is called Method Overriding. If we want to define a method in a base class that may be (but does not have to be) overridden by a derived class, we must mark the method with the **virtual** keyword. The virtual keyword is used to modify a method, property, indexer, or event declared in the base class and allow it to be overridden in the derived class.

When a derived class needs to change the implementation details of a virtual method, it does so by using the override keyword.

Sample Code:

```
class Employee
{
    float salary = 5000;
    //This method can now be "overridden" by a derived class.
    public virtual void GiveBonus(float bonusAmount)
    {
        salary += bonusAmount;
    }
    public virtual void PrintSalary()
    {
        Console.WriteLine(salary);
    }
}


class SalesPerson : Employee
{
    public override void GiveBonus(float bonusAmount)
    {
        float _bonusAmount = bonusAmount  * 2;
        base.GiveBonus(_bonusAmount);
    }
    public override void PrintSalary()
    {
        base.PrintSalary();
    }
}
```

```
class Manager : Employee
{
    public override void GiveBonus(float bonusAmount)
    {
        base.GiveBonus(bonusAmount);
    }
}

class Program
{
    static void Main(string[] args)
    {
        SalesPerson emp = new SalesPerson();
        emp.GiveBonus(500);
        emp.PrintSalary();
    }
 }
```

Each overridden method is free to leverage the default behavior using the base keyword. In this way, No need to completely reimplement the logic behind GiveBonus() but can reuse (and possibly extend) the default behavior of the base class.

Difference between Method overloading and Method overriding

Method Overloading	**Method Overriding**
It is an approach of defining multiple methods with the same name but with a different signature.	It is an approach of defining multiple methods with the same name and with the same signature.
Overloading a method can be performed within a base class or within the derived classes also.	Overriding of methods is not possible within the same class it must be performed under the derived classes.
To overload a base class method under the derived class, the derived class does not require permission from the base.	To override a base class method under the derived class, first, the derived class require explicit permission from its base.
This is all about defining multiple behaviors to a method.	This is all about changing the behavior of a method.
Used to implement static polymorphism.	Used to implement dynamic polymorphism.

No separate keywords are used to implement function overloading.	Use **virtual** keyword for base class function and **override** keyword in derived class function to implement function overriding.

Method Hiding

The process of hiding the implementation of the methods of a base class from the derived class is called Method Hiding. Using the **new** keyword in the derived class method, it hides the implementation of the base class method.

Sample Code:

```
class Circle
{
    public  void Area(double r)
    {
       Console.WriteLine(String.Format("Area of Circle : {0}",
       Math.PI * r * r));
    }
}
class Sphere : Circle
{
    public new void Area(double r)
    {
       Console.WriteLine(String.Format("Area of Sphere : {0}", 4 *
       Math.PI * r * r));
    }
}
class Program
{
    static void Main(string[] args)
    {
        Sphere C = new Sphere(); // Circle C = new Sphere();
        C.Area(4);
    }
}
```

In the above C# program, Base and Derived class both have Area() method implementation. On calling Area() on derived object will hide the base class Area method and call derived class method.

Use method hiding in C#:

If derived class want to implement some of methods of base class without disturbing base class.

If base class is available in a library that cannot be changed even if we want to do so intentionally. Use name hiding in C# then.

Overriding is when we provide a new override implementation of a method in a derived class when that method is defined in the base class as virtual.

Hiding is when we provide a new implementation of a method in a derived class when that method is not defined in the base class as virtual, or when our new implementation does not specify override.