**C#** **[See Sharp]**: C# |||2000||| is an object-oriented,***component-oriented*** and **type-safe** programming language|

**C# Used to** : create  robust and durable apps ex:Windows UI apps, backend services, controls, libraries, android apps, and even blockchain applications.

**C# features** : structured|||easy parameters passing||| different platform compilation|||Open-source|||Flexible and scalable ||

**C# Key Features** :

1. [***Garbage collection***](https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/) :automatically reclaims memory occupied by unreachable unused objects.
2. [***Nullable types***](https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references) guard against variables that don't refer to allocated objects.
3. [***Exception handling***](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/) provides a structured and extensible approach to error detection and recovery.
4. [***Lambda expressions***](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions) support functional programming techniques.
5. [***Language Integrated Query (LINQ)***](https://docs.microsoft.com/en-us/dotnet/csharp/linq/) syntax creates a common pattern for working with data from any source.
6. [***Asynchronous operations***](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/) provides syntax for building distributed systems.
7. C# has a [***unified type system***](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/). All C# types, including primitive types such as int and double, inherit from a sing

The history of C# evolves from C# 1.0 to C# version 11.0 is latest

**Class** : A class generally depicts{blueprint) the structure of data, how data is stored and managed within a program. It contains data members & member functions [properties, methods, & other objects ]

**Objects** : is the instance of a class or can say real-world entity of that particular class using which can have access to all the characteristics of that class.

**Types** : C# language are divided into two main categories: ***reference types*** and ***value types***. Both value types and reference types may be ***generic types***, which take one or more ***type parameters***.

**.NET architecture :** C# programs run on .NET, a virtual execution system called the common language runtime (CLR) and a set of class libraries.

**CLR** : is the implementation by Microsoft of the common language infrastructure (CLI), Source code written in C# is compiled into an [intermediate language (IL)](https://docs.microsoft.com/en-us/dotnet/standard/managed-code) that conforms to the CLI specification.

**Managed code :** Code that's executed by the CLR is sometimes referred to as **Managed code.**

**Unmanaged code** : Code that is compiled into native machine language that targets a specific platform is known as **Unmanaged code.**

**Assembly :** The IL code and resources, such as bitmaps and strings, are stored in an assembly, typically with an extension of *.dll*. it contains a manifest that provides information about the assembly's types, version, and culture.

**C# program is executed** : then the assembly is loaded into the CLR. The CLR performs Just-In-Time (JIT) compilation to convert the IL code to native machine instructions. The CLR provides other services related to **automatic garbage collection**, **exception handling,** and **resource management**.

**Key feature of .NET** : Language interoperability(IL code produced by the C# compiler conforms to the **Common Type Specification (CTS)**. IL code generated from C# can interact with code that was generated from the .NET versions of F#, Visual Basic, C++. There are more than 20 other CTS-compliant languages. A single assembly may contain multiple modules written in different .NET languages. The types can reference each other as if they were written in the same language.

.NET also includes **extensive libraries**. These libraries support many different workloads. The libraries include everything from file input and output to string manipulation to XML parsing, to web application frameworks to Windows Forms controls. The typical C# application uses the .NET class library extensively to handle common."

**C# Fundmentals :**

**T*ype*** : defines the structure and behavior of any data in C#,

**Variable** : is a label that refers to an instance of a specific type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Class type** defines a   * data structure that contains data members (fields) and function members (methods, properties, and others). * Supports single inheritance and polymorphism, mechanisms * whereby derived classes can extend and specialize base classes. | **Struct type** : is similar to a class type in that it represents a structure with data members and function members. However,  **unlike classes**,   * structs are value types. * **primary purpose** is to **store data values.** * **can't declare a base type.** * They implicitly derive from [System.ValueType](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype). * don't typically require heap allocation. * Struct types **don't support *user-specified inheritance***, * struct types implicitly inherit from type object. * **They're implicitly sealed.** | **Interface type**   * defines a contract as a named set of public members that can be implemented by classes and structs * An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces. * Interfaces may employ ***multiple inheritance***. | **Delegate type** :  **A delegate is a type-safe function pointer**   * **Represents references to methods** with a particular parameter list and return type. * Delegates make it possible to treat methods as entities that can be **assigned to variables and passed as parameters.** * Delegates similar to the concept of function pointers found in some other languages. * **Unlike function pointers, delegates are object-oriented and type-safe.** |

**Use of New** **Operator**: **Creates Instances of classes**, **which allocates memory for a new instance**, **invokes a constructor to initialize the instance**, and **returns a reference to the instance.**

|  |  |  |
| --- | --- | --- |
| * **Base class** : A class declaration may specify a base class. * Follow the class name and type parameters with a colon and the name of the base class. * Omitting(excluding) a base class specification is the same as deriving from type object. | * **Classes** are the most fundamental of C#'s types. * A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. * A class provides a definition for *instances* of the class, also known as *objects*. Classes support *inheritance* and *polymorphism*, mechanisms whereby *derived classes* can extend and specialize *base classes*. | **Generic classes : defines**[**type parameters**](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics)**.(ex:** **Pair<TFirst, TSecond>)**   * **A class type that is declared to take type parameters is called a *generic class type*.** * Type parameters are : a list of type parameter names enclosed in angle brackets. * Type parameters follow the class name. they can then be used in the body of the class. * Struct, interface, and delegate types can also be generic |

**Enums :** An [***Enum***](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum) type defines a set of constant values.

**Nullable types :** Variables of any type may be declared as ***non-nullable*** or ***nullable***. A nullable variable can hold an additional null value.

**Tuples:** C# supports [***tuples***](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples), which provides concise syntax to **group multiple data elements in a lightweight data structure**.

We cn instantiate a tuple by declaring the types and names of the members between ( and ) small brackets. Ex : (double Sum, int Count) t2 = (4.5, 3);

**Stack and Heap both are memory types.**

|  |  |
| --- | --- |
| **STACK like series of compartment i.e. one above another**   * **Keeps track of your running memory needed by your application.** * **Stack memory are value types like integer , double , Boolean etc** * **Stack is the running memory , the byval memory , So it just gets claimed when the application exits** | **HEAP  pile of objects which are kept one above another.**   * **They can be reached any moment of time.** * **Heap  memory are objects and string data types.** * **The by ref memory or heap memory is not claimed as soon as the application exits , but its claimed by the garbage collector routine.** |

**What do you mean by saying a "class is a reference type"?**

A class is a reference type means when an object of the class is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data**.**

**What do you mean by saying a "struct is a value type"?**

A struct is a value type mean when a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable therefore contain two separate copies of the same data. Changes made to one copy do not affect the other copy.

**C# program building blocks**

|  |  |  |
| --- | --- | --- |
| [**Members**](https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/program-building-blocks#members) **[ properties, fields, methods, & events.]** | [**Expressions**](https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/program-building-blocks#expressions) | * [Statements](https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/program-building-blocks#statements) |
| |  |  | | --- | --- | | **Static** | **Instance** | | Static members belong to classes, | Instance members belong to objects |  * **Constants**: Constant values associated with the class * **Fields**: Variables that are associated with the class * **Methods**: Actions that can be performed by the class * **Properties**: Actions associated with reading and writing named properties of the class * **Indexers**: Actions associated with indexing instances of the class like an array * **Events**: Notifications that can be generated by the class * **Operators**: Conversions and expression operators supported by the class * **Constructors**: Actions required to initialize instances of the class or the class itself * **Finalizers**: Actions done before instances of the class are permanently discarded * **Types**: Nested types declared by the class | * *Expressions* are constructed from *operands* and *operators*. * Examples of operators include +, -, \*, /, and new. * Examples of operands include literals, fields, local variables, and expressions. * The assignment operators, the null-coalescing ?? and ??= * conditional operator ?: are *right-associative*, meaning that operations are performed from right to left. | The actions of a program are expressed using *statements*.   * **Block** ***statements***: consists of a list of statements written between the delimiters { and }. * ***Declaration statements*** are used to declare local variables and constants. * ***Expression statements*** are used to evaluate expressions. * ***Selection statements*** are used to select one of a number of possible statements. * **Ex : if and switch statements.** * ***Iteration statements*** : are used to execute repeatedly an embedded statement. * **Ex : the while, do, for, and foreach statements.** * ***Jump statements*** are used to transfer control. **EX:** break, continue, goto, throw, return, and yield statements. * **The try...catch statement** : is used to catch exceptions that occur during execution of a block, and * **The try...finally statement** is used to specify finalization code that is always executed, whether an exception occurred or not. * **The checked and unchecked statements** : are used to control the overflow-checking context for integral-type arithmetic operations and conversions. * **The lock statement** : is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock. * **The using statement** : is used to obtain a resource, execute a statement, and then dispose of that resource. |

**Accessibility :** Each member of a class has an associated accessibility, which controls the regions of program text that can access the member.

There are six possible forms of accessibility. The access modifiers are summarized below.

* **public:** Access **isn't limited.**
* **private:** Access is **limited to this class**.
* **protected:** Access is limited to **this class** or **classes derived from this class**.
* **internal:** Access is limited to the **current assembly** (.exe or .dll).
* **protected internal:** Access is limited to **this class**, classes **derived from this class**, or **classes** **within the same assembly**.
* **private protected:** Access is limited **to this class** or classes **derived from this type within** **the same assembly.**
* **Parameters :** are used to pass values or variable references to methods.

|  |  |  |
| --- | --- | --- |
| * **A *Value parameter***  : Modifications to a value parameter don't affect the argument that was passed for the parameter. * Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted. | * **A *Reference parameter*** is used for passing arguments by reference. * The **argument passed for a reference parameter must be a variable with a definite value.** * During execution of the method, the **reference parameter represents the same storage location as the argument variable.** | * ***Output parameter*** is used for passing arguments by reference. * It's similar to a reference parameter, except that * it **doesn't require that you explicitly assign a value** to the caller-provided argument. * An output parameter is declared with the out modifier. * The syntax introduced in C# 7. |

|  |  |
| --- | --- |
| **Implicit conversions:**No special syntax is required because the.   1. conversion **is type safe and no data will be lost.** 2. conversions from **smaller to larger types, and** from **[derived classes to base classes]** 3. conversions from **Child to Parent.** 4. **Converstion is done automatically.** 5. **no data loss.** | **Explicit conversions : require a cast operator.**  **The source and destination variables are compatible, but there is a risk of data loss because the type of the destination variable is a smaller size than (or is a base class of) the source variable.** |

**Conversion : Converting a variable of one data type to another data type is called casting. This is also called as data type conversion.**

**What operators can be used to cast from one reference type to another without the risk of throwing an exception?**

**The is and as operators** can be used to **cast from one reference type to another without the risk of throwing an exception.**

**InvalidCastException** is thrown if casting failed.

|  |  |
| --- | --- |
| **Is operator** is used to   * **check the compatibility between two types or classes.** * **It returns a boolean value**. | **As operator** used to   * **perform conversion between compatible reference types or Nullable types**. * **Returns null if the two types or classes are not compatible with each other**. * Able to convert only if the 'to' type is less derived than the 'from' type |

**++i** means that when your code is executing **it will first do i = i + 1 and then read it.**

**i++** means that when your code is executing **it will first read it and do the i = i + 1**

**int** num

**Difference between Argument and Parameter**

|  |  |
| --- | --- |
| **int** num = 20;  Call(num)    // num is argument | **int** Call(**int** rnum)  {  **printf**("the num is %d", rnum); // rnum is parameter  } |

**What is delegate?**

Delegate is TYPE-SAFE function pointer. This means, delegate can point to any method as long as the method signature matches with that of delegate.

when we assign any method to **delegate**, we must keep in mind that **method should have matching parameters and return type.**

**What is Event?**

The event we define in one class is responsible to inform others that particular state has occurred, the audience is any class that subscribes to that defined event and ceremonies are operation performed by the classes which subscribed to that event.

Every event has two actors-

1. **Publisher** — Only one. Responsible to define the event, decide when to broadcast the event and broadcast the event at valid condition.
2. **Subscriber** — One or many. Responsible to listen to the event. Attach a method which will be called on occurrence of event and perform operations based on information passed through event.

**What is Reflection?**

The process of **obtaining information about assemblies** **and the types defined within them**, **and creating, invoking, and accessing type instances at run time**

**What is the purpose of async/await keywords?**

* These keywords allow writing **asynchronous non-blocking code in a synchronous fashion.**
* This feature is **facilitated by the Task/Task<T> classes or ValueTask/ValueTask<T> structs.**
* We use **await keyword to materialize the task into resulting** value.
* **Methods that contain await must be marked with the async keyword**.

**What's the difference between asynchronous programming and multithreaded programming?**

* asynchronous operation as **just an aggregation of two events** -- **start and finish.**
* It **does not represent execution on a separate thread.**

**Multithreaded execution *can be represented as an asynchronous operation but***

**Asynchronous operation necessarily not employs additional threads.**

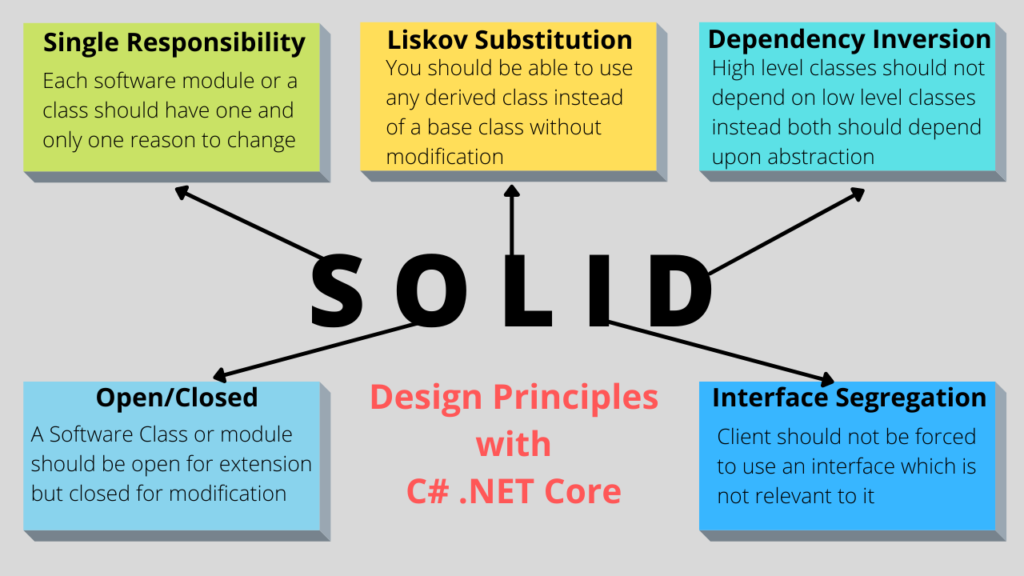
**Q: What happens if we execute an asynchronous method but don't await it?**

* Task will continue its lifetime normally, but the result won't be observed.
* Task object itself will eventually be reclaimed by garbage collector.
* Note\* : we can also use .ContinueWith() to handle the result in a call-back fashion.

**Q: What happens if an exception is thrown within an asynchronous method?**

If the method is awaited, the exception will instantly propagate to the calling method.

Otherwise, the exception will be considered unobserved,



|  |  |  |  |
| --- | --- | --- | --- |
| Inheritance |  |  |  |
| Inheritance is **a mechanism** is **a important Feature** using which we can reuse the Data members and member functions of the existing class. Inheritance facilitates Reusability. |  |  |  |

|  |  |  |
| --- | --- | --- |
| **Properties** | **Abstraction** | **Encapsulation** |
| Process or method of | gaining the information | contain the information |
| Problems are solved | design or interface level. | implementation level. |
| Method of | hiding the unwanted information | hide the data in a single entity  protect information from outside |
| Access | provides access to specific part of data | user can not access data directly |
| Focus is on | “what” should be done. | “How” it should be done. |

Ref vs Out

**Note : BY default value passed as byval.**

|  |  |  |
| --- | --- | --- |
| Params | Ref | Out |
| To pass variables to functions or methods by reference for | Yes | Yes |
| Pass data from caller to Calle with reference. | Yes | Yes |
| **Need to initialize inside caller function** | No | Yes [Must] |
| Two way | Yes | No |
| One way | No | Yes |