Classes consists of :

class members that include properties that describe class data, methods that define class behavior, and events that provide communication between different classes and objects

class SampleClass

{

private string \_sampleField;

public int SampleProperty { get; set; }

}

// Properties have get and set procedures, which provide more control on how values are set or returned.

// Fields are like variables because they can be read or set directly.

A struct type is a value type that is typically used to encapsulate small groups of related variables,

public struct Book

{

public decimal price;

public string title;

public string author;

}

Structs can also contain [constructors](https://msdn.microsoft.com/en-us/library/ace5hbzh.aspx), [constants](https://msdn.microsoft.com/en-us/library/ms173119.aspx), [fields](https://msdn.microsoft.com/en-us/library/ms173118.aspx), [methods](https://msdn.microsoft.com/en-us/library/ms173114.aspx), [properties](https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx), [indexers](https://msdn.microsoft.com/en-us/library/6x16t2tx.aspx), [operators](https://msdn.microsoft.com/en-us/library/ms173145.aspx), [events](https://msdn.microsoft.com/en-us/library/awbftdfh.aspx), and [nested types](https://msdn.microsoft.com/en-us/library/ms173120.aspx), although if several such members are required, you should consider making your type a class instead.

* Within a struct declaration, fields cannot be initialized unless they are declared as const or static.
* A struct cannot declare a default constructor (a constructor without parameters) or a destructor.
* Structs are copied on assignment. When a struct is assigned to a new variable, all the data is copied, and any modification to the new copy does not change the data for the original copy. This is important to remember when working with collections of value types such as Dictionary<string, myStruct>.
* Structs are value types and classes are reference types.
* Unlike classes, structs can be instantiated without using a new operator.
* Structs can declare constructors that have parameters.
* A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from System.ValueType, which inherits from System.Object.
* A struct can implement interfaces.
* A struct can be used as a nullable type and can be assigned a null value.

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type.

Class has internal access by default and if you specify it’s default constructor it is private by default otherwise in absence of default constructor it is public access.

A static constructor is used to initialize any [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced. If you have both instance and static constructor then static constructor will only fire when ever the static field or static class is referenced regardless of constructor call order which means either static constructor is called before instance or vice versa.

Constructor can be public,private,protected,internal,protected internal and static

Static constructors have the following properties:

* A static constructor does not take access modifiers or have parameters.
* A static constructor is called automatically to initialize the [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) before the first instance is created or any static members are referenced.
* ~~A static constructor cannot be called directly.~~
* ~~The user has no control on when the static constructor is executed in the program~~.
* If the class is never used, the **static constructor** is not guaranteed to be **called** at all.
* A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
* Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the LoadLibrarymethod.
* If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running.

Destructors

Destructors are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need destructors to clean up any unmanaged resources that your application creates. There can be only one destructor for a class.

Nested Classes

A class defined within another class is called *nested*. By default, the nested class is private.

C#

class Container

{

class Nested

{

// Add code here.

}

}

Need of nested class – more encapsulation and grouping similar functionality classes

To create an instance of the nested class, use the name of the container class followed by the dot and then followed by the name of the nested class:

C#

Container.Nested nestedInstance = new Container.Nested()

protected internalThe type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

To specify that a class cannot be used as a base class:

public sealed class A { }

To specify that a class can be used as a base class only and cannot be instantiated:

public abstract class B { }

The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

Abstract classes have the following features:

* An abstract class cannot be instantiated.
* An abstract class may contain abstract methods and accessors.
* It is not possible to modify an abstract class with the [sealed](https://msdn.microsoft.com/en-us/library/88c54tsw.aspx) modifier because the two modifers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
* A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Abstract methods have the following features:

* An abstract method is implicitly a virtual method.
* Abstract method declarations are only permitted in abstract classes.
* Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:
* public abstract void MyMethod();

The implementation is provided by an overriding method [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx), which is a member of a non-abstract class.

* It is an error to use the [static](https://msdn.microsoft.com/en-us/library/98f28cdx.aspx) or [virtual](https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx) modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

* It is an error to use the abstract modifier on a static property.
* An abstract inherited property can be overridden in a derived class by including a property declaration that uses the [override](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) modifier.

**Use an abstract class**  
  
-**When** creating a class library which will be widely distributed or reused—especially to clients, use an abstract class in preference to an interface; because, it simplifies versioning. This is the practice used by the Microsoft team which developed the Base Class Library. ( COM was designed around interfaces.)  
-**Use** an abstract class to define a common base class for a family of types.  
-**Use** an abstract class to provide default behavior.  
-**Subclass** only a base class in a hierarchy to which the class logically belongs.

**Use an interface**  
-When we do not have idea about implementation we should use intrerface.  
-**When** creating a standalone project which can be changed at will, use an interface in preference to an abstract class; because, it offers more design flexibility.

Lose coupling

Dependency Injection

Unit Testing and mocking  
-**Use** interfaces to introduce polymorphic behavior without subclassing and to model multiple inheritance—allowing a specific type to support numerous behaviors.  
-**Use** an interface to design a polymorphic hierarchy for value types.  
-**Use** an interface when an immutable contract is really intended.  
-**A well-designed interface** defines a very specific range of functionality. Split up interfaces that contain unrelated functionality.

Static(Shared in VB) class or Members are shared by all instance of a class

The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

public virtual double Area()

{

return x \* y;

}

<https://www.codeproject.com/Articles/18734/Method-Overriding-in-C>

The implementation of a virtual member can be changed by an [overriding member](https://msdn.microsoft.com/en-us/library/ebca9ah3.aspx) in a derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use **the virtual modifier with the static, abstract, private, or override modifiers.**

The override modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event

An override method provides a new implementation of a member that is inherited from a base class. The method that is overridden by an overridedeclaration is known as the overridden base method. The overridden base method must have the same signature as the override method.

**You cannot override a non-virtual or static method**. The overridden base method must be virtual, abstract, or override.

An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same [access level modifier](https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx).

You cannot use the new, static, or virtual modifiers to modify an override method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property, and the overridden property must be virtual, abstract, or override.

When used as a declaration modifier, the new keyword explicitly hides a member that is inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base class version. Although you can hide members without using the new modifier, you get a compiler warning. If you use new to explicitly hide a member, it suppresses this warning.

To hide an inherited member, declare it in the derived class by using the same member name, and modify it with the new keyword.

public class BaseC

{

public int x;

public void Invoke() { }

}

public class DerivedC : BaseC

{

new public void Invoke() { }

}

An interface contains definitions for a group of related functionalities that a [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx) or a [struct](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx) can implement.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

An interface has the following properties:

* An interface is like an abstract base class. Any class or struct that implements the interface must implement all its members.
* An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
* Interfaces can contain events, indexers, methods, and properties.
* Interfaces contain no implementation of methods.
* A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

Using interface based design concept provides loose coupling, component-based programming, easier maintainability, makes your code base more scalable and makes code reuse much more accessible because implementation is separated from the interface.

## **Delegates**

A delegate is a type that defines a method signature, and can provide a reference to any method with a compatible signature. You can invoke (or call) the method through the delegate. Delegates are used to pass methods as arguments to other methods.

Indexers are a syntactic convenience that enable you to create a [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx), [struct](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx), or [interface](https://msdn.microsoft.com/en-us/library/87d83y5b.aspx) that client applications can access just as an array. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array.

Generics

Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.

## **Generics Overview**

* Use generic types to maximize code reuse, type safety, and performance.
* The most common use of generics is to create collection classes.
* The .NET Framework class library contains several new generic collection classes in the [System.Collections.Generic](https://msdn.microsoft.com/en-us/library/system.collections.generic.aspx) namespace. These should be used whenever possible instead of classes such as [ArrayList](https://msdn.microsoft.com/en-us/library/system.collections.arraylist.aspx) in the [System.Collections](https://msdn.microsoft.com/en-us/library/system.collections.aspx) namespace.
* You can create your own generic interfaces, classes, methods, events and delegates.
* Generic classes may be constrained to enable access to methods on particular data types.
* Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Generics provide a code template for creating type-safe code without referring to specific data types. Generics allow you to realize type safety at compile time. They allow you to create a data structure without committing to a specific data type. When the data structure is used, however, the compiler ensures that the types used with it are consistent for type safety.

The event keyword is used to declare an event in a publisher class.

Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

### **Uses for Serialization**

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

### **Binary Serialization**

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams.

### **XML Serialization**

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. [System.Xml.Serialization](https://msdn.microsoft.com/en-us/library/system.xml.serialization.aspx) contains the classes necessary for serializing and deserializing XML.

### **Basic Serialization**

The only requirement in basic serialization is that the object has the [SerializableAttribute](https://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx) attribute applied. The [NonSerializedAttribute](https://msdn.microsoft.com/en-us/library/system.nonserializedattribute.aspx) can be used to keep specific fields from being serialized.

### **Custom Serialization**

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked [SerializableAttribute](https://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx) and implement the [ISerializable](https://msdn.microsoft.com/en-us/library/system.runtime.serialization.iserializable.aspx) interface.

<https://www.codeproject.com/Articles/22787/Custom-Serialization-Example>

Func is probably most commonly used in LINQ - for example in projections:

list.Select(x => x.SomeProperty)

Action is more commonly used for things like List<T>.ForEach: execute the given action for each item in the list.

Predicate is mostly used in List<T> for methods like FindAll and RemoveAll

**Reflection** provides objects (of type [Type](https://msdn.microsoft.com/en-us/library/system.type.aspx)) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.

Reflection is useful in the following situations:

* When you have to access attributes in your program's metadata. For more information, see [Retrieving Information Stored in Attributes](https://msdn.microsoft.com/en-us/library/71s1zwct.aspx).
* For examining and instantiating types in an assembly.
* For building new types at runtime. Use classes in [System.Reflection.Emit](https://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx).
* For performing late binding, accessing methods on types created at run time.

Typeof(string).IsClass – true

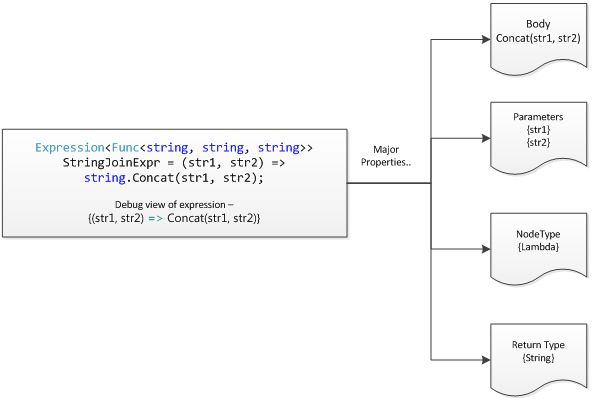
Typeof(int).IsValueType- true

Typeof(abc).IsGenricType – true

"The expression tree is an in-memory data representation of the lambda expression. The expression tree makes the structure of the lambda expression transparent and explicit. You can interact with the data in the expression tree just as you can with any other data structure."

Expression and Expression<> are basically classes that can represent the CSharp code as Data. Expression can be parsed, analyzed in the program.

1. Expression<Func<**string**, **string**, **string**>> StringJoinExpr = (str1, str2) => **string**.Concat(str1, str2);



1. ***// Create an expression using expression lambda***
2. **Expression<Func<int, int, int>> expression = (num1, num2) => num1 + num2;**
4. ***// Compile the expression***
5. **Func<int, int, int> compiledExpression = expression.Compile();**
7. ***// Execute the expression.***
8. **int result = compiledExpression(3, 4); *//return 7***

A more derived class object's array (String []) is being assigned to the less derived class object array variable (object []). This is called covariance in array.

object[] objArray = new String[10];

The Contravariance Delegates reverse the Covariance functionality. It allows a method that has parameter types less derived than what is specified in the delegate.

static void SetObject(object objectParameter) { }  
static void SetString(string stringParameter) { }  
static void Main()  
{  
    Action<string> del2 = SetObject;  
    .....  
}

LINQ

It enables you to query the data from the various data sources like SQL databases, XML documents, ADO.NET Datasets, Web services and any other objects such as Collections, Generics etc. by using a **SQL Query like syntax** with .NET framework languages like C# and VB.NET.

System.Linq namespace

Query Expression

IEnumerable<int> query =

from num in numbers

where num > 5 && num < 10

select num;

Method Invocation

IEnumerable<int> query = numbers.Where(num => num > 5 && num < 10);

Anonymous Type:

DataContext context = new DataContext();

var result = from book in context.Books

where book.Price > 200

orderby book.IssueDate descending

select new

{

Name = book.Name,

IssueNumber = "#" + book.Issue

};

Conversion Operators : AsEnumerable, Cast, ToList, ToArray and ToDictionary.

Element Operators : Single, SingleOrDefault, First, FirstOrDefault, Last, LastOrDefault.

**Single -** It returns a single specific element from a collection of elements if element match found. An exception is thrown, if none or more than one match found for that element in the collection.

**SingleOrDefault -** It returns a single specific element from a collection of elements if element match found. An exception is thrown, if more than one match found for that element in the collection. A default value is returned, if no match is found for that element in the collection.

**First -** It returns first specific element from a collection of elements if one or more than one match found for that element. An exception is thrown, if no match is found for that element in the collection.

**FirstOrDefault -** It returns first specific element from a collection of elements if one or more than one match found for that element. A default value is returned, if no match is found for that element in the collection.

From groupby where orderby select , no having

**Into -** This is used to store the results of a group, join or select clause into a temporary variable. It hides the previous range variable and create a temporary range variable which you can be used further. DataContext context = new DataContext();

var q=from emp in context.tblEmployee

group emp by new { emp.Salary, emp.EmpId }

into groupingEmp

let avgsalary = (groupingEmp.Sum(gEmp => gEmp.Salary) / groupingEmp.Count())

where groupingEmp.Key.Salary == avgsalary

select new { groupingEmp.Key.Salary, groupingEmp.Key.EmpId };

**Let -** This is used to store the result of a sub expression into a new variable. It doesn’t hide the previous range variable and create a new variable which can be used further with previous variable

**Deferred Execution:** In case of differed execution, a query is not executed at the point of its declaration. It is executed when the Query variable is iterated by using loop like as for, foreach. DataContext context = new DataContext();

var query = from customer in context.Customers

where customer.City == "Delhi"

select customer; // Query does not execute here

foreach (var Customer in query) // Query executes here

{

Console.WriteLine(Customer.Name);

}

**Immediate Execution:** In case of immediate execution, a query is executed at the point of its declaration. The query which returns a **singleton** value (a single value or a set of values) like Average, Sum, Count, List etc. caused Immediate Execution.

|  |
| --- |
| You can force a query to execute immediately of by calling ToList, ToArray methods. DataContext context = new DataContext();  var query = (from customer in context.Customers  where customer.City == "Delhi"  select customer).Count(); // Query execute here |

**Lazy/Deferred Loading:** In case of lazy loading, related objects (child objects) are not loaded automatically with its parent object until they are requested. By default LINQ supports lazy loading.

DataContext context = new DataContext();

var query = context.Department.Take(3); // Lazy loading

foreach (var Dept in query)

{

Console.WriteLine(Dept.Name);

foreach (var Emp in Dept.Employee)

{

Console.WriteLine(Emp.EmpID);

}

}

**Eager loading:** In case of eager loading, related objects (child objects) are loaded automatically with its parent object. To use Eager loading you need to use **Include**() method.

DataContext context = new DataContext();

var query = context.Department.Include("Employee").Take(3); // Eager loading

foreach (var Dept in query)

{

Console.WriteLine(Dept.Name);

foreach (var Emp in Dept.Employee)

{

Console.WriteLine(Emp.EmpID);

}

}

Inner Join :

var q = (from pd in context.Products

join od in context.Orders on pd.ProductID equals od.ProductID

orderby od.OrderID

select new

{

od.OrderID,

pd.ProductID,

pd.Name,

pd.UnitPrice,

od.Quantity,

od.Price,

}).ToList();

**GROUP JOIN**- When a join clause use an INTO expression, then it is called a group join

var q = (from pd in context.tblProducts

join od in context.tblOrders on pd.ProductID equals od.ProductID

into t

orderby pd.ProductID

select new

{

pd.ProductID,

pd.Name,

pd.UnitPrice,

Order = t

}).ToList();

Left Outer Join :

var q = (from pd in context.tblProducts

join od in context.tblOrders on pd.ProductID equals od.ProductID into t

from rt in t.DefaultIfEmpty()

orderby pd.ProductID

select new

{

OrderID = rt.OrderID,

pd.ProductID,

pd.Name,

pd.UnitPrice,

rt.Quantity,

rt.Price,

}).ToList();

**CROSS JOIN**- Cross join is a Cartesian join means Cartesian product of both the tables. This join does not need any condition to join two tables

var q = from c in context.Customers

from o in context.Orders

select new

{

c.CustomerID,

c.ContactName,

o.OrderID,

o.OrderDate

};

Having – No having clause in LINQ

var q=from ord in context.tblOrder

group ord by ord.CustomerID into grouping

where grouping.Count()>=2

select

new

{

grouping.Key,

grouping

};

**1.** Stored procedures are faster as compared to LINQ query since they have a predictable execution plan and can take the full advantage of SQL features. Hence, when a stored procedure is being executed next time, the database used the cached execution plan to execute that stored procedure.

**2.** LINQ has full **type checking** at compile-time and **IntelliSense** support in Visual Studio as compared to stored procedure. This powerful feature helps you to avoid run-time errors.

**3.** LINQ allows debugging through .NET debugger as compared to stored procedure.

**4.** LINQ also supports various .NET framework features like multi –threading as compared to stored procedures.

**5.** LINQ provides the uniform programming model (means common query syntax) to query the multiple databases while you need to re-write the stored procedure for different databases.

**6. Stored procedure is a best way for writing complex queries as compared to LINQ.**

**7.** Deploying LINQ based application is much easy and simple as compared to stored procedures based. Since in case of stored procedures, you need to provide a SQL script for deployment but in case of LINQ everything gets complied into the DLLs. Hence you need to deploy only DLLs.

1.LINQ query is compiled each and every time while stored procedures re-used the cached execution plan to execute. Hence, LINQ query takes more time in execution as compared to stored procedures.

**2. LINQ is not the good for writing complex queries as compared to stored procedures.**

**3. LINQ is not a good way for bulk insert and update operations.**

**4. Performance is degraded if you don't write the LINQ query correctly.**

**5. If you have done some changes in your query, you have to recompile it and redeploy its dll to the server.**

**1. KeepCurrentValues** - This option will remains the LINQ object values as it is and does not push the new values from the database in to the LINQ object.

**2. OverwriteCurrentValues** - This option will replace the LINQ object values with the database values.

**3. KeepChanges** - In this case changed properties of an object/entity remains as it is but the properties which are not changed are fetched from the database and replaced.

All these options are available in ***RefereshMode*** enum which exist in *System.Data.Linq* namespace.

|  |
| --- |
| To handle concurrency conflicts, wrap the code with in a try block and catch the *ChangeConflictException* by using catch block and iterate through the *ChangeConflicts* collection to resolve the conflict as shown below:  DataContext db = new DataContext();  try  {  // TO DO:  db.SubmitChanges(ConflictMode.ContinueOnConflict);  }  catch (ChangeConflictException ex)  {  foreach (ObjectChangeConflict changeconf in db.ChangeConflicts)  {  changeconf.Resolve(RefreshMode.OverwriteCurrentValues);  }  } |

In LINQ to SQL, you can also handle concurrency at field level and this is the best way provided by the LINQ. To achieve this you need to define ***UpdateCheck*** attribute at field level and it has the following options:

**1. Never: -** This option will never check for concurrency conflicts.

**2. Always: -** This option will always check for concurrency conflicts.

**3. WhenChanged: -** This option will check for concurrency conflicts when the field’s value has been changed.

|  |
| --- |
| [Column(DbType = "nvarchar(50)", UpdateCheck = UpdateCheck.Never)]  public string CustomerID  {  set{CustomerID = value;}  get{return \_CustomerID;}  } |

DataContext class acts as a bridge between SQL Server database and the LINQ to SQL. For accessing the database and also for changing the data in the database, it contains connections string and the functions.

Tuples are used to return multiple values from a function

Multithreading  
  
Threads have the following properties:

* Threads enable your program to perform concurrent processing.
* The .NET Framework [System.Threading](https://msdn.microsoft.com/en-us/library/system.threading(v=vs.100).aspx) namespace makes using threads easier.
* Threads share the application's resources. For more information, see [Using Threads and Threading](https://msdn.microsoft.com/en-us/library/e1dx6b2h(v=vs.100).aspx).

System.Threading.Thread newThread = new System.Threading.Thread(AMethod);

Calling the constructor, providing the name of the procedure or method that you want to execute on the new thread.

newThread.Start()

|  |  |
| --- | --- |
| **Method** | **Action** |
| [Start](https://msdn.microsoft.com/en-us/library/system.threading.thread.start(v=vs.100).aspx) | Causes a thread to start to run. |
| [Sleep](https://msdn.microsoft.com/en-us/library/system.threading.thread.sleep(v=vs.100).aspx) | Pauses a thread for a specified time. |
| [Suspend](https://msdn.microsoft.com/en-us/library/system.threading.thread.suspend(v=vs.100).aspx) | Pauses a thread when it reaches a safe point. |
| [Abort](https://msdn.microsoft.com/en-us/library/system.threading.thread.abort(v=vs.100).aspx) | Stops a thread when it reaches a safe point. |
| [Resume](https://msdn.microsoft.com/en-us/library/system.threading.thread.resume(v=vs.100).aspx) | Restarts a suspended thread |
| [Join](https://msdn.microsoft.com/en-us/library/system.threading.thread.join(v=vs.100).aspx) | Causes the current thread to wait for another thread to finish. If used with a time-out value, this method returns **True** if the thread finishes in the allocated time. |

Safe points are locations in code where it is safe for the common language runtime to perform automatic *garbage collection*, the process of releasing unused variables and reclaiming memory. When you call the [Abort](https://msdn.microsoft.com/en-us/library/system.threading.thread.abort(v=vs.100).aspx) or [Suspend](https://msdn.microsoft.com/en-us/library/system.threading.thread.suspend(v=vs.100).aspx) method of a thread, the common language runtime analyzes the code and determines the location of an appropriate location for the thread to stop running.

Thread States:

* Aborted -> Aborted already.
* AbortRequested -> Responding to an Abort() request.
* Background -> Running in background. Same as IsBackground property.
* Running -> Running after another thread has called the start()
* Stopped -> After finishing run() or Abort() stopped it.
* Suspended -> Suspended after Suspend() is called.
* Unstarted -> Created but start() has not been called.
* WaitSleepJoin -> Sleep()/Wait() on itself and join() on another thread. If a thread Thread1 calls sleep() on itself and calls join() on the thread Thread2 then it enters WaitSleepJoin state. The thread exists in this state till the timeout expires or another thread invokes Interrupt() on it.

### **Thread Properties**

Threads also contain several useful properties, as shown in the following table:

|  |  |
| --- | --- |
| **Property** | **Value** |
| [IsAlive](https://msdn.microsoft.com/en-us/library/system.threading.thread.isalive(v=vs.100).aspx) | Contains the value **True** if a thread is active. |
| [IsBackground](https://msdn.microsoft.com/en-us/library/system.threading.thread.isbackground(v=vs.100).aspx) | Gets or sets a Boolean that indicates if a thread is or should be a background thread. Background threads are like foreground threads, but a background thread does not prevent a process from stopping. Once all foreground threads that belong to a process have stopped, the common language runtime ends the process by calling the [Abort](https://msdn.microsoft.com/en-us/library/system.threading.thread.abort(v=vs.100).aspx) method on background threads that are still alive. |
| [Name](https://msdn.microsoft.com/en-us/library/system.threading.thread.name(v=vs.100).aspx) | Gets or sets the name of a thread. Most frequently used to discover individual threads when you debug. |
| [Priority](https://msdn.microsoft.com/en-us/library/system.threading.thread.priority(v=vs.100).aspx) | Gets or sets a value that is used by the operating system to prioritize thread scheduling. |
| [ApartmentState](https://msdn.microsoft.com/en-us/library/system.threading.thread.apartmentstate(v=vs.100).aspx) | Gets or sets the threading model used for a particular thread. Threading models are important when a thread calls unmanaged code. |
| [ThreadState](https://msdn.microsoft.com/en-us/library/system.threading.thread.threadstate(v=vs.100).aspx) | Contains a value that describes a thread's state or states. |

## **Thread Priorities**

Every thread has a priority property that determines how big or small a slice of processor time it has to execute. The operating system allocates longer time slices to high-priority threads and shorter time slices to low-priority threads. New threads are created with the value of **Normal**, but you can change the [Priority](https://msdn.microsoft.com/en-us/library/system.threading.thread.priority(v=vs.100).aspx) property to any value in the [ThreadPriority](https://msdn.microsoft.com/en-us/library/system.threading.threadpriority(v=vs.100).aspx) enumeration.

### **Members**

|  |  |  |
| --- | --- | --- |
|  | **Member name** | **Description** |
| Supported by the XNA Framework | Lowest | The [Thread](https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.100).aspx) can be scheduled after threads with any other priority. |
| Supported by the XNA Framework | BelowNormal | The [Thread](https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.100).aspx) can be scheduled after threads with **Normal** priority and before those with **Lowest** priority. |
| Supported by the XNA Framework | Normal | The [Thread](https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.100).aspx) can be scheduled after threads with **AboveNormal** priority and before those with **BelowNormal** priority. Threads have **Normal** priority by default. |
| Supported by the XNA Framework | AboveNormal | The [Thread](https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.100).aspx) can be scheduled after threads with **Highest** priority and before those with **Normal** priority. |
| Supported by the XNA Framework | Highest | The [Thread](https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.100).aspx) can be scheduled before threads with any other priority. |

The **lock** (C#) and **SyncLock** (Visual Basic) statements can be used to ensure that a block of code runs to completion without interruption by other threads. This is accomplished by obtaining a mutual-exclusion lock for a given object for the duration of the code block.

public class TestThreading

{

private System.Object lockThis = new System.Object();

public void Process()

{

lock (lockThis)

{

// Access thread-sensitive resources.

}

}

}

Generally, it is best to avoid locking on a **public** type, or on object instances beyond the control of your application. For example, lock(this) can be problematic if the instance can be accessed publicly, because code beyond your control may lock on the object as well. This could create deadlock situations where two or more threads wait for the release of the same object. Locking on a public data type, as opposed to an object, can cause problems for the same reason. Locking on literal strings is especially risky because literal strings are *interned* by the common language runtime (CLR). This means that there is one instance of any given string literal for the entire program, the exact same object represents the literal in all running application domains, on all threads. As a result, a lock placed on a string with the same contents anywhere in the application process locks all instances of that string in the application. As a result, it is best to lock a private or protected member that is not interned. Some classes provide members specifically for locking.

## **Monitors**

Like the **lock** and **SyncLock** keywords, monitors prevent blocks of code from simultaneous execution by multiple threads. The [Enter](https://msdn.microsoft.com/en-us/library/system.threading.monitor.enter(v=vs.100).aspx) method allows one and only one thread to proceed into the following statements; all other threads are blocked until the executing thread calls [Exit](https://msdn.microsoft.com/en-us/library/system.threading.monitor.exit(v=vs.100).aspx)

//This is code fired at background of lock process

System.Object obj = (System.Object)x;

System.Threading.Monitor.Enter(obj);

try

{

DoSomething();

}

finally

{

System.Threading.Monitor.Exit(obj);

}

**Monitor** has the following features:

* It is associated with an object on demand.
* It is unbound, which means it can be called directly from any context.
* An instance of the **Monitor** class cannot be created.

## **Mutex Object**

A *mutex* is similar to a monitor; it prevents the simultaneous execution of a block of code by more than one thread at a time. In fact, the name "mutex" is a shortened form of the term "mutually exclusive." Mutex ensures the thread safety which are out process that is threads which coming from outside of an application (External threads). Mutex provides safety against the external threads. In a multiple instance of an application external threads are created so to ensure thread safety from an external threads we need to apply mutex.

Semaphore

Semaphore is also helps us to work with external threads and identifying whether an application is acquired by an external thread or not. But unlike mutex Semaphore allows one or more threads to enter to executes their task with thread safety.

Using a lock or monitor is useful for preventing the simultaneous execution of thread-sensitive blocks of code, but these constructs do not allow one thread to communicate an event to another. This requires *synchronization events*, which are objects that have one of two states, signaled and un-signaled, that can be used to activate and suspend threads. Threads can be suspended by being made to wait on a synchronization event that is unsignaled, and can be activated by changing the event state to signaled.

There are two kinds of synchronization events: [AutoResetEvent](https://msdn.microsoft.com/en-us/library/system.threading.autoresetevent(v=vs.100).aspx), and [ManualResetEvent](https://msdn.microsoft.com/en-us/library/system.threading.manualresetevent(v=vs.100).aspx). They differ only in that [AutoResetEvent](https://msdn.microsoft.com/en-us/library/system.threading.autoresetevent(v=vs.100).aspx) changes from signaled to unsignaled automatically any time it activates a thread. Conversely, a [ManualResetEvent](https://msdn.microsoft.com/en-us/library/system.threading.manualresetevent(v=vs.100).aspx) allows any number of threads to be activated by its signaled state, and will only revert to an unsignaled state when its [Reset](https://msdn.microsoft.com/en-us/library/system.threading.eventwaithandle.reset(v=vs.100).aspx) method is called.

Threads can be made to wait on events by calling one of the wait methods, such as [WaitOne](https://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitone(v=vs.100).aspx), [WaitAny](https://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitany(v=vs.100).aspx), or [WaitAll](https://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitall(v=vs.100).aspx). [WaitHandle.WaitOne()](https://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitone(v=vs.100).aspx) causes the thread to wait until a single event becomes signaled, [WaitHandle.WaitAny()](https://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitany(v=vs.100).aspx) blocks a thread until one or more indicated events become signaled, and [WaitHandle.WaitAll()](https://msdn.microsoft.com/en-us/library/system.threading.waithandle.waitall(v=vs.100).aspx) blocks the thread until all of the indicated events become signaled. An event becomes signaled when its [Set](https://msdn.microsoft.com/en-us/library/system.threading.eventwaithandle.set(v=vs.100).aspx) method is called.

 The [ReaderWriterLockSlim](https://msdn.microsoft.com/en-us/library/system.threading.readerwriterlockslim(v=vs.100).aspx) class supports scenarios that require a single writer and multiple readers. The class enforces exclusive access to a resource while a thread is modifying the resource, but it allows non-exclusive access when reading the resource.

A thread pool is a collection of threads that can be used to perform a number of tasks in the background. (See [Using Threading](https://msdn.microsoft.com/en-us/library/5xt1dysy(v=vs.80).aspx) for background information.) This leaves the primary thread free to perform other tasks asynchronously.

Thread pools are often employed in server applications. Each incoming request is assigned to a thread from the thread pool, so the request can be processed asynchronously, without tying up the primary thread or delaying the processing of subsequent requests.

The Task Parallel Library (TPL) is based on the concept of a task, which represents an asynchronous operation. The term task parallelism refers to one or more independent tasks running concurrently. Tasks provide two primary benefits:

* More efficient and more scalable use of system resources.

Behind the scenes, tasks are queued to the [ThreadPool](https://msdn.microsoft.com/en-us/library/system.threading.threadpool(v=vs.110).aspx), which has been enhanced with algorithms that determine and adjust to the number of threads and that provide load balancing to maximize throughput. This makes tasks relatively lightweight, and you can create many of them to enable fine-grained parallelism.

* More programmatic control than is possible with a thread or work item.

Tasks and the framework built around them provide a rich set of APIs that support waiting, cancellation, continuations, robust exception handling, detailed status, custom scheduling, and more.

You can also use the [Task.Run](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.run(v=vs.110).aspx) methods to create and start a task in one operation.

Task taskA = Task.Run( () => Console.WriteLine("Hello from taskA."));

You can also use the [TaskFactory.StartNew](https://msdn.microsoft.com/en-us/library/system.threading.tasks.taskfactory.startnew(v=vs.110).aspx) method to create and start a task in one operation

Task taskA = Task.Factory.StartNew(() => Console.WriteLine("Hello from taskA."));

There are three return types, you can use in async methods.

1. Task  
2. Task<T>  
3. Void

Types of Task Exception

TaskCanceledException: A task was canceled.

// NotSupportedException: Specified method is not supported.

An async method will be run synchronously if it does not contain the await keyword.

### **Creating And Starting New Task**

1. **static** **void** Main(**string**[] args)
2. {
3. //creating the task
4. Task<**int**> task1 = **new** Task<**int**>(() =>
5. {
6. **int** result = 1;
8. **for** (**int** i = 1; i < 10; i++)
9. result \*= i;
11. **return** result;
12. });
14. //starting the task
15. task1.Start();
17. //waiting for result - printing to the console
18. Console.WriteLine("Task result: {0}", task1.Result);
20. Console.WriteLine("Main method complete. Press any key to finish.");
21. Console.ReadKey();
22. }

### **Cancelling A Task**

CancellationTokenSource cancellationTokenSource = **new** CancellationTokenSource();

1. CancellationToken token = cancellationTokenSource.Token;
2. cancellationTokenSource.Cancel();
3. **static** **void** Main(**string**[] args)
4. {
5. //creating the cancelation token
6. CancellationTokenSource cancellationTokenSource = **new** CancellationTokenSource();
7. CancellationToken token = cancellationTokenSource.Token;
9. //creating the task
10. Task task = **new** Task(() =>
11. {
12. **for** (**int** i = 0; i < 100000; i++)
13. {
14. **if** (token.IsCancellationRequested)
15. {
16. Console.WriteLine("Cancel() called.");
17. **return**;
18. }
20. Console.WriteLine("Loop value {0}", i);
21. }
22. }, token);
24. Console.WriteLine("Press any key to start task");
25. Console.WriteLine("Press any key again to cancel the running task");
26. Console.ReadKey();
28. //starting the task
29. task.Start();
31. //reading a console key
32. Console.ReadKey();
34. //canceling the task
35. Console.WriteLine("Canceling task");
36. cancellationTokenSource.Cancel();
38. Console.WriteLine("Main method complete. Press any key to finish.");
39. Console.ReadKey();
40. }

C# Latest Features

## Throw expressions

It is easy to throw an exception in the middle of an expression: just call a method that does it for you! But in C# 7.0 we are directly allowing throw as an expression in certain places:

class Person

{

public string Name { get; }

public Person(string name) => Name = name ?? throw new ArgumentNullException(name);

public string GetFirstName()

{

var parts = Name.Split(" ");

return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");

}

public string GetLastName() => throw new NotImplementedException();

}

## Literal improvements

C# 7.0 allows \_ to occur as a **digit separator** inside number literals:

var d = 123\_456;

var x = 0xAB\_CD\_EF;

* 1. Out Variables

 the ability to declare a variable right at the point where it is passed as an out argument:

public void PrintCoordinates(Point p)

{

p.GetCoordinates(out int x, out int y);

WriteLine($"({x}, {y})");

}

### 4. Switch statements with patterns

We’re generalizing the switch statement so that:

* You can switch on any type (not just primitive types)
* Patterns can be used in case clauses
* Case clauses can have additional conditions on them

switch(shape)

{

case Circle c:

WriteLine($"circle with radius {c.Radius}");

break;

case Rectangle s when (s.Length == s.Height):

WriteLine($"{s.Length} x {s.Height} square");

break;

case Rectangle r:

WriteLine($"{r.Length} x {r.Height} rectangle");

break;

default:

WriteLine("<unknown shape>");

break;

case null:

throw new ArgumentNullException(nameof(shape));

}