

Collections

- It contains a set of classes to contain elements in a generalized manner.
- With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.
- C# divide collection in several classes. It contains `System.Collections.Generic`, `System.Collection` and `System.Collections.Concurrent`.

System.Collections.Generic Classes

- It provides a generic implementation of standard data structure like linked lists, stacks, queues, and dictionaries.
- These collections are type-safe because that are type-compatible with the type of the collection can be stored in a generic collection.
- Generic collections are defined by the set of interfaces and classes

Class name	Description
Dictionary<TKey,TValue>	It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.
List<T>	It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class.

List<T>

- The generic List class is the most commonly used.
- The generic List class provides a dynamically sized array of objects and are among the most commonly used of the collection classes.
- Whereas List<T> implements both IList and IList<T>.

```
List<string> words = new List<string>(); // New string-typed list
```

```
    words.Add ("melon");
```

```
    words.Add ("avocado");
```

System.Collections Classes

- It is a general-purpose data structure that works on object references, so it can handle any type of object, but not in a safe-type manner.
- Non-generic collections are defined by the set of interfaces and classes.

Class name	Description
ArrayList	It is a dynamic array means the size of the array is not fixed, it can increase and decrease at runtime.
Hashtable	It represents a collection of key-and-value pairs that are organized based on the hash code of the key.

Delegate

- Delegates are similar to pointers to functions, in C or C++.
- A delegate is an object which refers to a method.
- It is a reference type variable that can hold a reference to the methods. The reference can be changed at runtime.
- A delegate type defines the kind of method that delegate instances can call.
- Specifically, it defines the method's return type and its parameter types.
- Delegates are especially used for implementing events and the call-back methods.

Declaration Of Delegates

- Delegate type can be declared using the delegate keyword.
- Once a delegate is declared, delegate instance will refer and call those methods whose return type and parameter-list matches with the delegate declaration.

Syntax:

```
[access_modifier] delegate [return_type] [delegate_name] (parameter_type,  
parameter_name);
```

Example:


```
public delegate int operation(int x, int y);
```

- A delegate object is created with the help of **new** keyword.

```
[delegate_name] [instance_name] = new [delegate_name](calling_method_name);
```

```
using System;  
namespace Delegates
```

```
{  
    public delegate int operation(int x, int y);  
    class Program  
    {  
        static int Addition(int a, int b)  
        {  
            return a + b;  
        }  
        static void Main(string[] args)  
        {  
            operation obj = new operation(Program.Addition);  
            Console.WriteLine("Addition is={0}",obj(23,27));  
            Console.ReadLine();  
        }  
    }  
}
```



Multicast Delegates

- All delegate instances have multicast capability.
- This means that a delegate instance can reference not just a single target method, but also a list of target methods.
- The + and += operators combine delegate instances.

For example:

```
SomeDelegate d = SomeMethod1;
```

```
d += SomeMethod2;
```

This is similar with : `d = d + SomeMethod2;`

Invoking d will now call both SomeMethod1 and SomeMethod2.

Delegates are invoked in the order they are added.

CONTD...

- The - and -= operators remove the right delegate operand from the left delegate operand. For example:

```
d -= SomeMethod1;
```

Invoking d will now cause only SomeMethod2 to be invoked.

- Delegates are immutable, so when you call += or -=, you're in fact creating a new delegate instance and assigning it to the existing variable.

Advantages

- A delegate design may be a better choice than an interface design if one or more of these conditions are true:
- The interface defines only a single method.
- Multicast capability is needed.
- The subscriber needs to implement the interface multiple times

Events

- Events enable a class or object to notify other classes or objects when something of interest occurs.
- The class that sends (or raises) the event is called the publisher and the classes that receive (or handle) the event are called subscribers.
- The publishers determines when an event is raised and the subscriber determines what action is taken in response.
- The subscribers are the method target recipients.
- A subscriber decides when to start and stop listening, by calling += and -= on the broadcaster's delegate.
- A subscriber does not know about, or interfere with, other subscribers.

CONTD...

To declare an event inside a class, first of all, you must declare a delegate type for the event as:

```
public delegate string MyEventHandler(string str);
```

Syntax for the declaration of Event

- `public event MyEventHandler MyEvent;`

Implementing Event

- To declare an event inside a class, first a Delegate type for the Event must be declared like below:

```
public delegate void MyEventHandler(object sender, EventArgs e);
```

```
namespace SampleApp {  
    public delegate string MyDel(string str);  
    class EventProgram {  
        event MyDel MyEvent;  
        public EventProgram() {  
            this.MyEvent += new MyDel(this.WelcomeUser);  
        }  
        public string WelcomeUser(string username) {  
            return "Welcome " + username;  
        }  
        static void Main(string[] args) {  
            EventProgram obj1 = new EventProgram();  
            string result = obj1.MyEvent("Event call");  
            Console.WriteLine(result);  
        } }  
}
```


Steps To Choose Or Define A Delegate For The Event

There are three rules:

- It must have a void return type.
 - It must accept two arguments: the first of type object, and the second a subclass of EventArgs.
 - The first argument indicates the event broadcaster, and the second argument contains the extra information to convey.
 - Its name must end with EventHandler
- The Framework defines a generic delegate called System.EventHandler<> that satisfies these rules:

```
public delegate void EventHandler (object source, EventArgs e) where EventArgs :  
EventArgs;
```

Event Accessors

- An event's accessors are the implementations of its += and -= functions.
- By default, accessors are implemented implicitly by the compiler.

```
public event EventHandler PriceChanged;
```

The compiler converts this to the following:

- A private delegate field
- A public pair of event accessor functions (add_PriceChanged and remove_PriceChanged), whose implementations forward the += and -= operations to the private delegate field

File IO

- File and stream I/O (input/output) refers to the transfer of data either to or from a storage medium.
- In .NET, the System.IO namespaces contain types that enable reading and writing, both synchronously and asynchronously, on data streams and files.
- These namespaces also contain types that perform compression and decompression on files, and types that enable communication through pipes and serial ports.
- A file is an ordered and named collection of bytes that has persistent storage.
- When you work with files, you work with directory paths, disk storage, and file and directory names.
- The System.IO namespace to interact with files and directories.

Some Commonly Used File And Directory Classes

- **File** - provides static methods for creating, copying, deleting, moving, and opening files, and helps create a `FileStream` object.
- **FileInfo** - provides instance methods for creating, copying, deleting, moving, and opening files, and helps create a `FileStream` object.
- **Directory** - provides static methods for creating, moving, and enumerating through directories and subdirectories.
- **DirectoryInfo** - provides instance methods for creating, moving, and enumerating through directories and subdirectories.
- **Path** - provides methods and properties for processing directory strings in a cross-platform manner.

LINQ

- LINQ, or Language Integrated Query, is a set of language and framework features for writing structured type-safe queries over local object collections and remote data sources.
- LINQ enables us to query any collection implementing `IEnumerable<T>`, whether an array, list, or XML DOM, as well as remote data sources, such as tables in a SQL Server database.
- LINQ offers the benefits of both compile-time type checking and dynamic query composition.
- All core types are defined in the `System.Linq` and `System.Linq.Expressions` namespaces.
- The basic units of data in LINQ are sequences and elements.

CONTD...

- A sequence is any object that implements `IEnumerable<T>` and an element is each item in the sequence.
- A query operator is a method that transforms a sequence. A typical query operator accepts an input sequence and emits a transformed output sequence.
- Queries that operate over local sequences are called local queries or LINQ-to-objects queries.
- A query is an expression that, when enumerated, transforms sequences with query operators.
- The simplest query comprises one input sequence and one operator

LINQ Query Syntax:

from <range variable> in <IEnumerable<T> or IQueryable<T> Collection>

<Standard Query Operators> <lambda expression>

<select or groupBy operator> <result formation>

CONTD...

```
string[] names = { "Ram", "Hari", "Gopal" };
```

- **Using extension methods and lambda expressions.**

```
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where (names, n =>  
n.Length >= 4);
```

OR

```
IEnumerable filteredNames = names.Where (n => n.Contains ("a"));  
foreach (string name in filteredNames)  
    Console.WriteLine (name);
```

CONTD...

- **Using query expression**

```
IEnumerable<string> filteredNames = from n in names
```

```
    where n.Contains ("a")
```

```
    select n;
```

Query Expressions

- C# provides a syntactic shortcut for writing LINQ queries, called query expressions.
- The design of query expressions was inspired primarily by list comprehensions from functional programming languages such as LISP and Haskell, although SQL had a cosmetic influence.
- Query expressions always start with a **from** clause and end with either a **select** or **group clause**.
- The from clause declares a range variable (in this case, n), which we can think of as traversing the input sequence
- Another syntax for writing queries, query expression syntax

```
IEnumerable filteredNames = from n in names  
                             where n.Contains ("a")  
                             select n;
```


Chaining Query Operators

- To build more complex queries, we append additional query operators to the expression, creating a chain.

```
using System;
```

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

```
using System.Linq;
```

```
class LinqDemo {
```

```
    static void Main() {
```

```
        string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

```
        IEnumerable<string> query = names
```

```
        .Where (n => n.Contains ("a"))
```

```
        .OrderBy (n => n.Length)
```

```
        .Select (n => n.ToUpper());
```

```
        foreach (string name in query)
```

```
            Console.WriteLine (name);
```

```
    }
```

```
}
```

LINQ Query Syntax

```

IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
};

var teenAgerStudent = from s in studentList
    where s.Age > 12 && s.Age < 20
    select s;
```

CONTD...

```
var teenAgerStudent = from s in studentList
```

```
    where s.Age > 12 && s.Age < 20
```

```
        order by s.Name
```

```
    select s;
```

- `OrderBy(p => p.Name, StringComparer.CurrentCultureIgnoreCase);`

Using Lamda Expression

- `var studentList1 = studentsStream`
 `.Where<Student>(stu => stu.Age > 15)`
 `.OrderBy(p => p.Name);`
- `var studentList2 = studentsStream.Where(s => s.Age > 15).ToList<Student>();`

Query Syntax Versus SQL Syntax

- Query expressions look superficially like SQL, yet the two are very different.
- A LINQ query boils down to a C# expression, and so follows standard C# rules.
- For example, with LINQ, we cannot use a variable before we declare it.
- In SQL, we can reference a table alias in the SELECT clause before defining it in a FROM clause.
- A subquery in LINQ is just another C# expression and so requires no special syntax.
- Subqueries in SQL are subject to special rules.
- With LINQ, data logically flows from left to right through the query.
- With SQL, the order is less well-structured with regard data flow.

CONTD...

- A LINQ query comprises a conveyor belt or pipeline of operators that accept and emit sequences whose element order can matter.
- A SQL query comprises a network of clauses that work mostly with unordered sets.

Advantages Of LINQ

- LINQ offers an object-based, language-integrated way to query over data no matter where that data came from. So through LINQ we can query database, XML as well as collections.
- Compile time syntax checking
- It allows you to query collections like arrays, enumerable classes etc

Types Of LINQ

- LINQ to objects
- LINQ to SQL(DLINQ)
- LINQ to dataset
- LINQ to XML(XLINQ)
- LINQ to entities

Implementing The Enumeration Interfaces

- We might want to implement **IEnumerable** or **IEnumerable<T>** for one or more of the following reasons:
 - To support the foreach statement
 - To interoperate with anything expecting a standard collection
 - To meet the requirements of a more sophisticated collection interface
 - To support collection initializers

CONTD...

- To implement **IEnumerable/IEnumerable<T>**, you must provide an enumerator.

We can do this in one of three ways:

- If the class is “wrapping” another collection, by returning the wrapped collection’s enumerator
- Via an iterator using yield return
- By instantiating your own IEnumerator/IEnumerator<T> implementation

Lambda Expressions

- A lambda expression is an unnamed method that can be used to create delegates.
- There are two types of lambda expression, Expression Lambda and Statement Lambdas..
- *A lambda expression is used* to create an anonymous function.
- The compiler immediately converts the lambda expression to either:
 - A delegate instance.
 - An expression tree, of type `Expression<TDelegate>`, representing the code inside the lambda expression in a traversable object model.
 - This allows the lambda expression to be interpreted later at runtime.
- Internally, the compiler resolves lambda expressions of this type by writing a private method, and moving the expression's code into that method.
- The '`=>`' is the lambda operator. A lambda expression has the following form:
(input-parameters) => expression-or-statement-block

Lambda Expressions Can Be Of Two Types

Expression Lambda: Consists of the input and the expression.

- It is a type of lambda that has an expression to the right of the lambda operator.
- Syntax: **input => expression;**

number => (number % 2 == 0)

Statement Lambda: Consists of the input and a set of statements to be executed.

- It is a statement lambda because it contains a statement block {...} to the right side of the expression.
- Syntax: **input => { statements };**

number => { return number > 5 }

CONTD...

- Given the following delegate type: `delegate int Transformer (int i);`

we could assign and invoke the lambda expression `x => x * x` as follows:

```
Transformer sqr = x => x * x;
```

```
Console.WriteLine (sqr(3)); // 9
```

- Lambda expressions are used most commonly with the `Func` and `Action` delegates.
- `Func sqr = x => x * x;`
- Here's an example of an expression that accepts two parameters:

```
Func<string,string,int> totalLength = (s1, s2) => s1.Length + s2.Length;
```

```
int total = totalLength ("hello", "world"); // total is 10;
```

```
using System;
using System.Collections.Generic;
using System.Linq;
public static class demo
{
    public static void Main()
    {
        List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
        List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);
        foreach (var num in evenNumbers)
        {
            Console.Write("{0} ", num);
        }
        Console.WriteLine("press any key to exit");
        Console.Read();
    }
}
```



```
using System;
using System.Collections.Generic;
using System.Linq;
class Student {
    public int rollNo { get; set; }
    public string name { get; set; }
}
class LamdaDemo {
    static void Main(string[] args)
    {
        List<Student> details = new List<Student>() {
            new Student{ rollNo = 1, name = "Liza" }, new Student{ rollNo = 2, name = "Gita" },
            new Student{ rollNo = 3, name = "Tina" }, new Student{ rollNo = 4, name = "Sita" }
        };
        var newDetails = details.OrderBy(x => x.name);
        foreach(var value in newDetails)
        {
            Console.WriteLine(value.rollNo + " " + value.name);
        }
    }
}
```

Lambda Expressions Versus Local Methods

- The functionality of local methods overlaps with that of lambda expressions.
- Local methods have the following three advantages:
 - They can be recursive (they can call themselves), without ugly hacks
 - They avoid the clutter of specifying a delegate type
 - They incur slightly less overhead
- Local methods are more efficient because they avoid the indirection of a delegate (which costs some CPU cycles and a memory allocation).
- In many cases we need a delegate, most commonly when calling a higher-order function, i.e., a method with a delegate-typed parameter:

```
public void Foo (Func<int,bool> predicate) { ... }
```

Exception Handling

- An exception is a problem that arises during the execution of a program.
- A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
- exception handling deals with any unexpected or exceptional situations that occur when a program is running.
- exception handling is done by upon four keywords: **try**, **catch**, **finally**, and **throw**.

CONTD...

try

- A try block identifies a block of code for which particular exceptions is activated.
- The try block must be followed by a catch block, a finally block, or both.
- It is followed by one or more catch blocks.

catch

- The catch block executes when an error occurs in the try block.
- A program catches an exception with an exception handler at the place in program where you want to handle the problem.
- The catch keyword indicates the catching of an exception.
- **finally** –
 - A finally block always executes—whether or not an exception is thrown and whether or not the try block runs to completion.
 - finally blocks are typically used for cleanup code.

CONTD...

- The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown.
- For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** – A program throws an exception when a problem shows up.

This is done using a throw keyword.


```
try
{
    ...// exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
    ...// handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
    ...// handle exception of type ExceptionB
}
finally
{
    ...// cleanup code
}
```

CONTD...

```
class Test
{
    static int Calc (int x) => 10 / x;
    static void Main()
    {
        int y = Calc (0);
        Console.WriteLine (y);
    }
}
```

- Because x is zero, the runtime throws a `DivideByZeroException`, and program terminates.
- We can prevent this by catching the exception as follows:

```
class Test {  
    static int Calc (int x) => 10 / x;  
    static void Main() {  
        try  
        {    int y = Calc (0);  
            Console.WriteLine (y);  
        }  
        catch (DivideByZeroException ex)  
        {  
            Console.WriteLine ("x cannot be zero");  
        }  
        Console.WriteLine ("program completed");  
    }  
}
```

OUTPUT: x cannot be zero

program completed

Throwing Exceptions

- Exceptions can be thrown either by the runtime or in user code.
- In this example, Display throws a `System.ArgumentNullException`:

```
class Test {  
  
    static void Display (string name) {  
  
        if (name == null)  
  
            throw new ArgumentNullException (nameof (name));  
  
            Console.WriteLine (name);  
  
    }  
  
    static void Main() {  
  
        try { Display (null); }  
  
        catch (ArgumentNullException ex)  
  
        {  
  
            Console.WriteLine ("Caught the exception"); }  
  
        }  
  
    }
```


Rethrowing An Exception

- we can capture and rethrow an exception as follows:

```
try { ... }  
catch (Exception ex)  
{  
    // Log error  
    throw; // Rethrow same exception  
}
```

Common Exception Types

- The following exception types are used widely throughout the CLR and .NET Framework.
- we can throw these or use them as base classes for deriving custom exception types
 - `System.ArgumentException`
 - `System.ArgumentNullException`
 - `System.ArgumentOutOfRangeException`
 - `System.InvalidOperationException`
 - `System.NotSupportedException`
 - `System.NotImplementedException`
 - `System.ObjectDisposedException`

Attribute

- C# enables programmers to invent new kinds of declarative information, called attributes.
- Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment.

Attribute Classes

- A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an attribute class.
- The declaration of an attribute class defines a new kind of attribute that can be placed on program entities.
- By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.
- An attribute defines additional information that is associated with a class, structure, method, and so on.
- Attributes are specified between square brackets, preceding the item to which they apply.
- An attribute specifies supplemental information that is attached to an item.

Attribute Usage

- The attribute `AttributeUsage` is used to describe how an attribute class can be used.
- `AttributeUsage` has a positional parameter that enables an attribute class to specify the kinds of program entities on which it can be used.
- Defines an attribute class named `SimpleAttribute` that can be placed on **`class_declarations`** and **`interface_declarations`** only.

Example:

```
using System;
```

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
```

```
public class SimpleAttribute :Attribute
```

```
{  
    }  
}
```


Example

[Simple]

```
class ClassI {...}
```

[Simple]

```
interface InterfaceI {...}
```

Positional And Named Parameters

- Attribute classes can have **positional parameters** and **named parameters**.
- Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class.
- Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class.
- For a property to define a named parameter, that property shall have both a public get accessor and a public set accessor.

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute : Attribute
{
    string url;
    public HelpAttribute(string url) // url is a positional parameter
    {
        this.url = url;
    }
    public string Topic { get; set; } // Topic is a named parameter
    public string Url {
        get
        {
            return url;
        }
    }
}
```

CONTD...

- Defines an attribute class named **HelpAttribute** that has one positional parameter, **url**, and one named parameter, **Topic**.
- Although it is non-static and public, the property **Url** does not define a named parameter, since it is not read-write.

Example:

```
[Help("http://www.mycompany.com/.../Class1.htm")]
```

```
class Class1 { }
```

```
[Help("http://www.mycompany.com/.../Misc.htm",Topic ="Class2")]
```

```
class Class2 { }
```

AttributeTargets

- Enumeration values can be combined with a bitwise OR operation to get the preferred combination.

```
using System;
```

```
namespace AttTargsCS
```

```
{ // This attribute is only valid on a class.
```

```
[AttributeUsage(AttributeTargets.Class)]
```

```
public class ClassTargetAttribute : Attribute
```

```
{ }
```


Specifying Multiple Attributes

- Multiple attributes can be specified for a single code element.
- Each attribute can be listed within the same pair of square brackets (separated by a comma), in separate pairs of square brackets, or in any combination of the two.
- Consequently, the following three examples are semantically identical:

[Serializable, Obsolete, CLSCompliant(false)]

public class Bar {...}

[Serializable]

[Obsolete]

[CLSCompliant(false)]

public class Bar {...}

-
- [Serializable, Obsolete]
 - [CLSCompliant(false)]
 - public class Bar {...}