Introduction to C#

Contents

[Struct Type 2](#_Toc441733055)

[Class Type 2](#_Toc441733056)

[Interface Type 4](#_Toc441733057)

[Array Type 5](#_Toc441733058)

[Generic Type 5](#_Toc441733059)

[Implicit Type 6](#_Toc441733060)

[Anonymous Type 6](#_Toc441733061)

[Nullable Type 7](#_Toc441733062)

[Value Parameters 8](#_Toc441733063)

[Reference Parameter 8](#_Toc441733064)

[Named Parameters 9](#_Toc441733065)

[Optional Parameters 9](#_Toc441733066)

[Delegate Type (1) 10](#_Toc441733067)

[Lambda Expressions 10](#_Toc441733068)

[Syntax of Lambda Expressions 11](#_Toc441733069)

[Examples: 11](#_Toc441733070)

[Data Types 11](#_Toc441733071)

[Extension Methods 11](#_Toc441733072)

[LINQ 12](#_Toc441733073)

[Filtering Operators 13](#_Toc441733074)

[Sorting Operators 13](#_Toc441733075)

[Set 14](#_Toc441733076)

[Quantifiers 14](#_Toc441733077)

[Projection 14](#_Toc441733078)

[Partitioning 15](#_Toc441733079)

[Joining 15](#_Toc441733080)

[Grouping 16](#_Toc441733081)

[Generators 16](#_Toc441733082)

[Equality 16](#_Toc441733083)

[Element 16](#_Toc441733084)

[Conversions 16](#_Toc441733085)

[Concatentation 16](#_Toc441733086)

[Aggregation 16](#_Toc441733087)

# Struct Type

|  |
| --- |
| using System;  namespace StructType  {  public struct Coordinate  {  public int X, Y;  public Coordinate(int p1, int p2)  {  X = p1;  Y = p2;  }  }  class Program  {  private static void Main(string[] args)  {  Coordinate c1;  c1.X = 10;  c1.Y = 20;  Console.WriteLine("x = {0}, y = {1}", c1.X, c1.Y);  Coordinate c2 = new Coordinate(10, 10);  Console.WriteLine("x = {0}, y = {1}", c2.X, c2.Y);  }  }  } |
| x = 10, y = 20  x = 10, y = 10  c1: x = 10, y = 20  c3: x = 4, y = 20 |

# Class Type

|  |
| --- |
| using System;  namespace ClassType1  {  public class Coordinate  {  public int X, Y;  public Coordinate()  {  X = 0;  Y = 0;  }  public Coordinate(int p1, int p2)  {  X = p1;  Y = p2;  }    }  class Program  {  private static void Main()  {  Coordinate c1 = new Coordinate {X = 10, Y=20};  Console.WriteLine("c1: x = {0}, y = {1}", c1.X, c1.Y);  Coordinate c2 = new Coordinate(10, 10);  Console.WriteLine("c2: x = {0}, y = {1}", c2.X, c2.Y);  Coordinate c3 = c1;  c3.X = 4;  Console.WriteLine("c1: x = {0}, y = {1}", c1.X, c1.Y);  Console.WriteLine("c3: x = {0}, y = {1}", c3.X, c3.Y);  }  }  } |
| c1: x = 10, y = 20  c2: x = 10, y = 10  c1: x = 4, y = 20  c3: x = 4, y = 20 |

|  |
| --- |
| using System;  namespace ClassType2  {  public class Employee  {  private string name = "NoName";  public string Name  {  get { return name; }  set { name = value; }  }  }  public class Manager : Employee  {  public string Id { get; set; }  public string Shout()  {  return "GET BACK TO WORK!";  }  }  class Program  {  private static void Main()  {  Employee e = new Employee {Name = "Jeff"};  Manager m = new Manager {Id = "1234", Name = "Evelyn"};  Console.WriteLine("'{0} {1}' said {2}", e.Name, m.Shout(), m.Name);  }  }  } |
| 'Jeff GET BACK TO WORK!' said Evelyn |

# Interface Type

|  |
| --- |
| using System;  namespace InterfaceType  {  interface IArea  {  double GetArea();  }  public class Rectangle : IArea  {  public double Length { get; set; }  public double Width { get; set; }  public double GetArea()  {  return Length \* Width;  }  }  public class Circle : IArea  {  public double Radius { get; set; }  public double GetArea()  {  return Math.PI \* Radius \* Radius;  }  }  class Program  {  private static void Main()  {  IArea r = new Rectangle { Length = 20, Width = 10 };  ShowArea(r);  IArea c = new Circle { Radius = 10 };  ShowArea(c);  }  private static void ShowArea(IArea ia)  {  Console.WriteLine("The area of the shape is {0}", ia.GetArea());  }  }  } |
| The area of the shape is 200  The area of the shape is 314.159265358979 |

# Array Type

|  |
| --- |
| namespace ArrayType  {  class Program  {  private static void Main()  {  // Declare a single-dimensional array  int[] array1 = new int[5];  // Declare and set array element values  int[] array2 = new int[] { 1, 3, 5, 7, 9 };  // Alternative syntax  int[] array3 = { 1, 2, 3, 4, 5, 6 };  // Declare a two dimensional array  int[,] multiDimensionalArray1 = new int[2, 3];  // Declare and set array element values  int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };  // Declare a jagged array  int[][] jaggedArray = new int[6][];  // Set the values of the first array in the jagged array structure  jaggedArray[0] = new int[4] { 1, 2, 3, 4 };  }  }  } |

# Generic Type

|  |
| --- |
| namespace GenericType  {  // Declare the generic class.  public class GenericList<T>  {  void Add(T input) { }  }  public class SomeClass { }  class Program  {  private static void Main()  {  // Declare a list of type int.  GenericList<int> list1 = new GenericList<int>();  // Declare a list of type string.  GenericList<string> list2 = new GenericList<string>();  // Declare a list of type ExampleClass.  GenericList<SomeClass> list3 = new GenericList<SomeClass>();  }  }  } |

# Implicit Type

|  |
| --- |
| using System.Collections.Generic;  namespace ImplicitType  {  public class Program  {  private static void Main()  {  // i is compiled as an int  var i = 5;  // s is compiled as a string  var s = "Hello";  // a is compiled as int[]  var a = new[] { 0, 1, 2 };  // list is compiled as List<int>  var list = new List<int>();  }  }  } |

# Anonymous Type

|  |
| --- |
| using System;  namespace AnonType  {  public class Program  {  private static void Main()  {  var v = new { Amount = 108, Message = "Hello" };  Console.WriteLine(v.Amount + " " + v.Message);  var anonArray =  new[]  {  new { name = "apple", diam = 4, color="red" },  new { name = "grape", diam = 1, color="purple" }  };  foreach (var o in anonArray)  {  Console.WriteLine("Color={0}, Name={1}", o.color, o.name);  }  }  }  } |
| 108 Hello  Color=red, Name=apple  Color=purple, Name=grape |

# Nullable Type

|  |
| --- |
| namespace NullableType  {  public class Program  {  private static void Main()  {  int? x = 10;  if (x.HasValue)  {  System.Console.WriteLine(x.Value);  }  else  {  System.Console.WriteLine("Undefined");  }  int? c = null;  if (c.HasValue)  {  System.Console.WriteLine(c.Value);  }  else  {  System.Console.WriteLine("Undefined");  }  // d = c, unless c is null, in which case d = -1.  int d = c ?? -1;  System.Console.WriteLine(d);  int? e = null;  int? f = null;  // g = e or f, unless e and f are both null, in which case g = -1.  int g = e ?? f ?? -1;  System.Console.WriteLine(g);  }  }  } |
| 10  Undefined  -1  -1 |

# Value Parameters

|  |
| --- |
| namespace ValueParam  {  class Program  {  // The parameter x is passed by value.  // Changes to x will not affect the original value of x.  static void SquareIt(int x)  {  x \*= x;  System.Console.WriteLine("The value inside the method: {0}", x);  }  private static void Main()  {  int n = 5;  System.Console.WriteLine("The value before calling the method: {0}", n);  SquareIt(n); // Passing the variable by value.  System.Console.WriteLine("The value after calling the method: {0}", n);  }  }  } |
| The value before calling the method: 5  The value inside the method: 25  The value after calling the method: 5 |

# Reference Parameter

|  |
| --- |
| namespace ReferenceParam  {  class Program  {  // The parameter x is passed by reference.  // Changes to x will affect the original value of x.  static void SquareIt(ref int x)  {  x \*= x;  System.Console.WriteLine("The value inside the method: {0}", x);  }  private static void Main()  {  int n = 5;  System.Console.WriteLine("The value before calling the method: {0}", n);  SquareIt(ref n); // Passing the variable by reference.  System.Console.WriteLine("The value after calling the method: {0}", n);  }  }  } |
| The value before calling the method: 5  The value inside the method: 25  The value after calling the method: 25 |

# Named Parameters

|  |
| --- |
| using System;  namespace NamedParam  {  class Program  {  private static void Main()  {  // The method can be called in the normal way, by using positional arguments.  Console.WriteLine(CalculateBMI(123, 64));  // Named arguments can be supplied for the parameters in either order.  Console.WriteLine(CalculateBMI(weight: 123, height: 64));  Console.WriteLine(CalculateBMI(height: 64, weight: 123));  // Positional arguments cannot follow named arguments.  // The following statement causes a compiler error.  //Console.WriteLine(CalculateBMI(weight: 123, 64));  // Named arguments can follow positional arguments.  Console.WriteLine(CalculateBMI(123, height: 64));  }  private static int CalculateBMI(int weight, int height)  {  return (weight \* 703) / (height \* height);  }  }  } |
| 21  21  21  21 |

# Optional Parameters

|  |
| --- |
| public class ExampleClass  {  private readonly string \_name;  // Because the parameter for the constructor, name, has a default  // value assigned to it, it is optional.  public ExampleClass(string name = "Default name")  {  \_name = name;  }  // The first parameter, required, has no default value assigned  // to it. Therefore, it is not optional. Both optionalstr and  // optionalint have default values assigned to them. They are optional.  public void ExampleMethod(int required, string optionalstr = "default string",  int optionalint = 10)  {  Console.WriteLine("{0}: {1}, {2}, and {3}.", \_name, required, optionalstr,  optionalint);  }  } |

# Delegate Type (1)

* A class that can hold a reference to a method
* It is defined as a method signature (which includes the return type in this context)
* It allows methods to be passed as parameters
* Can be chained together, e.g. one event can trigger multiple method calls
* Allows the use of anonymous methods and lambda expressions

|  |
| --- |
| namespace DelegateType  {  // Define the delegate type  public delegate void **Del**(string message);  public class Program  {  // Create a method that matches the delegate's signature  public static void DelegateMethod(string message)  {  System.Console.WriteLine(message);  }  private static void Main()  {  // Instantiate the delegate.  **Del** handler = DelegateMethod;  // Call the delegate.  handler("Hello, World!");  }  }  } |
| Hello, World! |

Lambda Expressions[[1]](#footnote-1)

A lambda expression is an anonymous function that you can use to write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator ‘=>,’ and you put the expression or statement block on the other side. For example, the lambda expression ‘x => x \* x’ specifies a parameter that’s named ‘x’ and returns the value of ‘x’ squared.

Syntax of Lambda Expressions

(input parameters) => expression

(input parameters) => {statements;}

Examples:

|  |  |
| --- | --- |
| (x) => x \* x | Returns |
| (x, y) => Math.Pow(x, y) | Returns |
| () =>  {  Console.WriteLine("Hello, World!");  }; | Takes no parameters and writes “Hello, World!” to the console. |
| (g) =>  {  string message = g + ", " + "World";  Console.WriteLine(message);  }; | Takes one parameter prepends it to “World” and then writes the message to the console. |

Data Types

|  |  |
| --- | --- |
| (x) => x \* x | **Func<double, double>**  A function that takes a ‘double’ parameter and returns a ‘double’.  *The last parameter type is the return type. There can be only one return type.* |
| (x, y) => Math.Pow(x, y) | **Func<double, double, double>**  A function that takes two ‘double’ parameters and returns a ‘double’. |
| () =>  {  Console.WriteLine("Hello, World!");  }; | **Action**  A function that takes no parameters and returns nothing. |
| (g) =>  {  string message = g + ", " + "World";  Console.WriteLine(message);  }; | **Action<string>**  A function that takes a string parameter and returns nothing. |

Extension Methods[[2]](#footnote-2)

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. For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

|  |
| --- |
| public **static class** StringExtensions  {  public **static** char MiddleChar(string myString)  {  return myString[myString.Length / 2];  }  } |
| string name = "Jeffrey";  Console.WriteLine(StringExtensions.MiddleChar(name)); |
| ***Output***  f |

|  |
| --- |
| public **static class** StringExtensions  {  public **static** char MiddleChar(**this** string myString)  {  return myString[myString.Length / 2];  }  } |
| string name = "Jeffrey";  Console.WriteLine(name.MiddleChar()); |
| ***Output***  f |

LINQ[[3]](#footnote-3)

Language-Integrated Query (LINQ) is a set of features introduced in Visual Studio 2008 that extends powerful query capabilities to the language syntax of C# and Visual Basic. LINQ introduces standard, easily-learned patterns for querying and updating data, and the technology can be extended to support potentially any kind of data store. Visual Studio includes LINQ provider assemblies that enable the use of LINQ with .NET Framework collections, SQL Server databases, ADO.NET Datasets, and XML documents.

Namespace: System.Linq

Most LINQ operators are extension methods defined on IEnumerable<T> and IQueryable<T>

Operators are either deferred or immediate

* Deferred Operators are either streaming or non-streaming. For example,
  + .Select() is streaming [It doesn't have to see the entire collection]
  + .OrderBy() is non-streaming [It has to see the entire collection]

|  |
| --- |
| List<Supplier> suppliers = new List<Supplier>  {  new Supplier {Id=1, Name="Acme" },  new Supplier {Id=2, Name="Zappo" },  new Supplier {Id=3, Name="Creepy" }  };  List<Part> parts = new List<Part>  {  new Part {Id=1, Name="Catapult", Price = 200.0m, SupplierId = 1 },  new Part {Id=2, Name="Zipper", Price = 2.0m, SupplierId = 2 },  new Part {Id=3, Name="Nail", Price = 12.0m, SupplierId = 1 },  new Part {Id=3, Name="Nail", Price = 8.0m, SupplierId = 2 },  new Part {Id=3, Name="Screw", Price = 8.0m, SupplierId = 2 }  };  int[] twos = { 0, 2, 4, 6, 8, 10, 12, 14 };  int[] threes = { 0, 3, 6, 9, 12, 15, 18, 21 }; |

Filtering Operators

Where() – Filters a sequence using a predicate

|  |
| --- |
| var q = parts.Where(p => p.Price < 12.0m);  foreach(var item in q)  {  Console.WriteLine(item.Name);  } |
| Zipper  Nail  Screw |

Sorting Operators

* OrderBy()
* OrderByDescending()
  + ThenBy()
* Reverse()

|  |
| --- |
| var q = parts.OrderBy(p => p.Price).ThenBy(p => p.Name);  foreach (var item in q)  {  Console.WriteLine(item.Name + " " + item.Price);  } |
| Zipper 2.0  Nail 8.0  Screw 8.0  Nail 12.0  Catapult 200.0 |

Set

* Distinct() - Removes duplicates
* Except() - The difference of two sequences
* Intersect() - The intersection of two sequences
* Union() - Unique elements from both sequences

|  |
| --- |
| var q1 = twos.Except(threes);  foreach (var item in q1)  {  Console.WriteLine(item);  }  Console.WriteLine();  var q2 = twos.Intersect(threes);  foreach (var item in q2)  {  Console.WriteLine(item);  } |
| 2  4  8  10  14  0  6  12 |

Quantifiers

* All() - true, if all elements satisfy a condition
* Any() - true, if any element satisfy a condition
* Contains() - true, if the sequence contains a specific element

Projection

* Select() - Projects values in a sequence based on a transformation function

|  |
| --- |
| var q = parts  .Select(p =>  new {  Name = p.Name,  OldPrice = p.Price,  NewPrice = p.Price \* 1.1m  });  foreach (var item in q)  {  Console.WriteLine(item.Name + " " + item.OldPrice + " " + item.NewPrice);  } |
| Catapult 200.0 220.00  Zipper 2.0 2.20  Nail 12.0 13.20  Nail 8.0 8.80  Screw 8.0 8.80 |

Partitioning

* Skip(), SkipWhile() - skip elements until a condition or predicate is met
* Take(), TakeWhile() - Take elements until a condition or predicate is met

Joining

* Join() - Join two sequences on a key and yields a sequence (flat)
* GroupJoin() - Join two sequences on a key and yields groups of sequences (hierarchical)

|  |
| --- |
| var q1 = suppliers  .Join(  parts,  s => s.Id,  p => p.SupplierId,  (s, p) => new { SupplierName = s.Name, PartName = p.Name }  );  foreach (var item in q1)  {  Console.WriteLine(item.SupplierName + " " + item.PartName);  }  Console.WriteLine();  var q2 = suppliers  .GroupJoin(  parts,  s => s.Id,  p => p.SupplierId,  (s, theParts) => new { SupplierName = s.Name, Parts = theParts }  );  foreach (var item in q2)  {  Console.WriteLine(item.SupplierName);  foreach(var part in item.Parts)  {  Console.WriteLine(" --> " + part.Name + " " + part.Price);  }  } |
| Acme Catapult  Acme Nail  Zappo Zipper  Zappo Nail  Zappo Screw  Acme  --> Catapult 200.0  --> Nail 12.0  Zappo  --> Zipper 2.0  --> Nail 8.0  --> Screw 8.0  Creepy |

Grouping

* GroupBy() - group elements from a sequence [Lazy]
* ToLookup() - inserts elements into a one to many dictionary [Greedy]

Generators

* Empty() - returns an empty collection
* Range() - Generates a sequence of numbers
* Repeat() - Generates a collection of repeated values
* DefaultIfEmpty() - Replaces empty collection with collection of 1 default value

Equality

* SequenceEqual() - true, if the elements in two sequences are the same objects in the same order

Element

* ElementAt() / ElementAtOrDefault() - returns the element at a specified index
* First() / FirstOrDefault() - returns the first element of a collection
* Last() / LastOrDefault() - returns the last element of a collection
* Single() / SingleOrDefault() - returns a single element

Conversions

* AsEnumerable() - Returns input as IEnumerable<T>
* AsQueryable() - Converts IEnumerable<T> to IQueryable<T>
* ToArray() - Converts sequence to an array
* ToDictionary() - Converts sequence to a Dictionary<K, V>
* ToList() - Converts sequence to List<T>
* ToLookUp() - Group elements into an IGrouping<K, V>

Concatentation

* Concat() - Concatenates two sequences into a single sequence

Aggregation

* Average() - Calculates the average value in a sequence
* Count() / LongCount() - Counts the elements in a sequence, may also accept a predicate
* Max() - Returns the maximum value in a sequence
* Min() - Returns the minimum value in a sequence

Sum() - Calculates the sum of values in a sequence

1. <https://msdn.microsoft.com/en-us/library/bb397687.aspx> [↑](#footnote-ref-1)
2. <https://msdn.microsoft.com/en-us/library/bb383977.aspx> [↑](#footnote-ref-2)
3. <https://msdn.microsoft.com/en-us/library/bb397926.aspx> [↑](#footnote-ref-3)