Introduction to C#

# Hello, World!

|  |
| --- |
| using System;  /\*  A Hello World! program in C#.  \*/  namespace HelloWorld  {  public class Hello  {  private static void Main()  {  Console.WriteLine("Hello, World!");  // Keeps the console window open in debug mode.  Console.WriteLine("Press any key to exit.");  Console.ReadKey();  }  }  } |

# Types

Value types

* simple types
* enum types
* struct types

Reference types

* class types
* interface types
* delegate types
* array types
* generic types

Implicit local types (var)

Anonymous types

Nullable types (e.g. int?)

# Type System Unification

All types derive from the type object (Object)

|  |  |
| --- | --- |
| Method | Description |
| public virtual bool **Equals**(Object) | Determines whether the specified object is equal to the current object. |
| public static bool **Equals**(Object, Object) | Determines whether the specified object instances are considered equal. |
| protected virtual void **Finalize**() | Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. |
| public virtual int **GetHashCode**() | Serves as the default hash function. |
| public Type **GetType**() | Gets the Type of the current instance. |
| protected Object **MemberwiseClone**() | Creates a shallow copy of the current Object. |
| public static bool **ReferenceEquals**(Object, Object) | Determines whether the specified Object instances are the same instance. |
| public virtual string **ToString**() | Returns a string that represents the current object. |

# Simple Types

|  |  |
| --- | --- |
| bool | System.Boolean |
| byte | System.Byte |
| sbyte | System.SByte |
| char | System.Char |
| decimal | System.Decimal |
| double | System.Double |
| float | System.Single |
| int | System.Int32 |
| uint | System.UInt32 |
| long | System.Int64 |
| ulong | System.UInt64 |
| short | System.Int16 |
| ushort | System.UInt16 |

# Enumeration Types

Defined in namespace or nested within a class or struct

|  |
| --- |
| using System;  namespace EnumType  {  public class Program  {  private enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };  private enum Months : byte { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };  private static void Main()  {  Days today = Days.Monday;  int dayNumber = (int)today;  Console.WriteLine("{0} is day number #{1}.", today, dayNumber);  Months thisMonth = Months.Dec;  byte monthNumber = (byte)thisMonth;  Console.WriteLine("{0} is month number #{1}.", thisMonth, monthNumber);  }  }  } |
| Monday is day number #1.  Dec is month number #11. |

# 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! |