Simple C# Tutorial

C# Tutorial

- Introducing the .NET framework
- Comparing C# to C++ and Java
- Getting Started
- Variable Types
- Arrays
- Operators
- Flow Control
- Introducing Classes, Structs and Namespaces
- Class Declaration
- Introducing Methods
- Polymorphism (Inherited Methods)
- Constants, Fields, Properties and Indexers
- Delegates and Events
- Exceptions
- Code Documentation

Introducing the Microsoft .NET Framework

- .NET (dot-net) is the name Microsoft gives to its general vision of the future of computing, the view being of a world in which many applications run in a distributed manner across the Internet.
- We can identify a number of different motivations driving this vision.
  - Object-oriented programming
  - Compiled once and run everywhere.
  - Service-oriented application
- .NET is Microsoft JVM?
- .NET has been built upon open standard technologies like XML and SOAP and is towards more open standards rather than Microsoft its proprietary tendencies.

Introducing the Microsoft .NET Framework

- At the development end of the .NET vision is the .NET Framework (Microsoft JDK?) that contains:
  - The Common Language Runtime,
  - The .NET Framework Classes, and
  - higher-level features like ASP.NET and WinForms for developing desktop applications.
- The Common Language Runtime (CLR) (Microsoft JRE?) manages the execution of code compiled for the .NET platform. The CLR has two features:
  - Its specification has been opened up so that it can be ported to non-Windows platforms.
  - Any number of different languages can be used to manipulate the .NET framework classes, and the CLR will support them.

C#

- Not all of the supported languages fit entirely neatly into the .NET framework, but the one language that is guaranteed to fit in perfectly is C#.
- C# (C Sharp), a successor to C++, has been released in conjunction with the .NET framework.
- C# design goals:
  - Be comfortable for C++ programmer
  - Fit cleanly into the .NET Common Language Runtime (CLR)
  - Simplify the C++ model
  - Provide the right amount of flexibility
  - Support component-centric development

C# versus Java (Similarity)

- C# and Java are both languages descended from C and C++.
- Each includes advanced features, like garbage collection, which remove some of the low level maintenance tasks from the programmer. In a lot of areas they are syntactically similar.
- Both C# and Java compile initially to an intermediate language:
  - C# to Microsoft Intermediate Language (MSIL), and Java to Java bytecode.
  - In each case the intermediate language can be run - by interpretation or just-in-time compilation - on an appropriate virtual machine. In C#, however, more support is given for the further compilation of the intermediate language code into native code.
- Like Java, C# gives up on multiple class inheritance in favor of a single inheritance model. C# supports the multiple inheritance of interfaces.

C# versus Java (Differences)

- C# contains more primitive data types than Java, and also allows more extension to the value types.
  - For example, C# supports enumerations, type-safe value types which are limited to a defined set of constant variables, and structs, which are user-defined value types .
  - Java doesn't have enumerations, but can specify a class to emulate them .
- Unlike Java, C# has the useful feature that we can overload various operators.
- However, polymorphism is handled in a more complicated fashion, with derived class methods either **overriding** or **hiding** super class methods.
- In Java, multi-dimensional arrays are implemented solely with single-dimensional arrays where arrays can be members of other arrays. In addition to **jagged arrays**, however, C# also implements genuine rectangular arrays.

C# versus C++ (Differences)

- C# uses **delegates** - type-safe method pointers. These are used to implement event-handling.
- Although it has some elements derived from Visual Basic and Java, C++ is C#'s closest relative.
- In an important change from C++, C# code does not require header files. All code is written inline.
- The .NET runtime in which C# runs performs memory management takes care of tasks like garbage collection. Because of this, the use of pointers in C# is much less important than in C++.
- Pointers can be used in C#, where the code is marked as **unsafe**, but they are only really useful in situations where performance gains are at an absolute premium.
- Generally speaking, all C# types is ultimately derived from the **object** type.

C# versus C++ (Differences)

- There are also specific differences in the way that certain common types can be used. For instance, C# arrays are bounds checked unlike in C++, and it is therefore not possible to write past the end of a C# array.
- C# statements are quite similar to C++ statements. To note just one example of a difference: the 'switch' statements has been changed so that 'fall-through' behavior is disallowed.
- As mentioned above, C# gives up on the idea of multiple class inheritance. Other differences relating to the use of classes are: there is support for class 'properties' of the kind found in Visual Basic, and class methods are called using the . operator rather than the :: operator.

Getting Started – Hello World!

- In order to use C# and the .NET framework classes, first install the .NET framework SDK.
- Write the C# code.

```
using System;
public class HelloWorld {
   public static void Main() {
      // This is a single line comment.
      /*
       * This is a multiple line comment.
       */
      Console.WriteLine("Hello World!");
   }
}
```


- To compile the program on Mono, use the command: mcs HelloWorld.cs (csc HelloWorld.cs in .NET framework SDK)
- To run the program on Mono: mono HelloWorld.exe

Variable Types

- C# is a type-safe language. Variables are declared as being of a particular type, and each variable is constrained to hold only values of its declared type.
- Variables can hold either **value types** or **reference types**, or they can be pointers.
- A variable of value types directly contains only an object with the value.
- A variable of reference type directly contains a reference to an object. Another variable many contain a reference to the same object.
- It is possible in C# to define your own value types by declaring **enumerations** or **structs**.

C# Pre-defined Value Types

C# Type	.Net Framework Type	Signed	Bytes	Possible Values
sbyte	System.sbyte	Yes	1	-128 to 127
short	System.Int16	Yes	2	-32768 to 32767
int	System.Int32	Yes	4	2^{31} to 2^{31} - 1
long	System.Int64	Yes	8	2^{63} to 2^{63} - 1
byte	System.Byte	No	1	0 to 255
ushort	System.Uint16	No	2	0 to 65535
uint	System.Uint32	No	4	0 to 2^{32} - 1
ulong	System.Uint64	No	8	0 to 2^{64} - 1
float	System.Single	Yes	4	±1.5 x 10-45 to ±3.4 x 1038 with 7 significant figures
double	System.Double	Yes	8	±5.0 x 10-324 to ±1.7 x 10308 with 15 or 16 significant figures
decimal	System.Decimal	Yes	12	±1.0 x 10-28 to ±7.9 x 1028 with 28 or 29 significant figures
char	System.Char	N/A	2	Any Unicode character
bool	System.Boolean	N/A	1/2	true or false

Value Types

- Value Types: int x = 10;
- Reference Types: New reference types can be defined using 'class', 'interface', and 'delegate' declarations

  object.

```
object x = new object();
x.myValue = 10;
```

- Escape Sequences and Verbatim Strings

```
string a = "\"Hello World\nHow are you\"";
```

- Boxing: C# allows you convert any value type to a corresponding reference type, and to convert the resultant 'boxed' type back again.

```
int i = 123;
object box = i;
if (box is int)
    {Console.Write("Box contains an int");} // this line is printed
```

Pointers

- A **pointer** is a variable that holds the memory address of another type. In C#, pointers can only be declared to hold the memory addresses of value types.
- Pointers are declared implicitly, using the **dereferencer** symbol *. The operator **&** returns the memory address of the variable it prefixes.
- Example: What is the value of i?

```
int i = 5;
int *p;
p = &i;
*p = 10;
```

- The use of pointers is restricted to code which is marked as **unsafe** (memory access).

Pointers

- To address the problem of garbage collection, one can declare a pointer within a **fixed** expression.
- Any value types declared within unsafe code are automatically fixed, and will generate compile-time errors if used within fixed expressions. The same is not true for reference types.
- Although pointers usually can only be used with value types, an exception to this involves arrays.
- A pointer can be declared in relation to an array, as in the following:

```
int[] a = {4, 5};
int *b = a;
```

What happens in this case is that the memory location held by b is the location of the first type held by a.

Arrays

- Single-dimensional arrays have a single dimension

```
int[] i = new int[100];
```

- C# supports two types of multidimensional arrays: rectangular and jagged.
  - A **rectangular array** is a multidimensional array that has the fixed dimensions' sizes.

```
int[,] squareArray = new int[2,3];
int[,] squareArray = {{1, 2, 3}, {4, 5, 6}};
```

  - A **jagged arrays** is a multidimensional array that has the irregular dimensions' sizes.

```
int[][] jag = new int[2][];
jag[0] = new int [4];
jag[1] = new int [6];
int[][] jag = new int[][] {new int[] {1, 2, 3, 4}, new int[] {5, 6, 7, 8, 9, 10}};
```

Enumerations

- An **enumeration** is a special kind of value type limited to a restricted and unchangeable set of numerical values.
- When we define an enumeration we provide literals which are then used as constants for their corresponding values. The following code shows an example of such a definition:

```
public enum DAYS { Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday, Sunday};
enum byteEnum : byte {A, B};
```

- Instead, the numerical values are set up according to the following two rules:
  - For the **first literal**: if it is unassigned, **set its value to 0**.
  - For **any other literal**: if it is unassigned, then **set its value to one greater than the value of the preceding literal**.

Enumerations

```
using System;
public class EnumTest {

  public enum DAYS: byte
    {Monday, Tuesday, Wednesday, Thursday, Friday,
    Saturday, Sunday};

  public static void Main() {
    Array dayArray =
    Enum.GetValues(typeof(EnumTest.DAYS));

    foreach (DAYS day in dayArray)
      Console.WriteLine("Number {1} of EnumTest.DAYS is {0}",
    day, day.ToString("d"));
  }
}
```

Enumerations

- Console.WriteLine("Number {1} of EnumTest.DAYS is {0}", day, day.ToString("d"))

  is equivalent to:

  Console.WriteLine(String.Format("Number {1} of EnumTest.DAYS is {0}", day, day.ToString("d")));
- And what the String.Format method does is to take **textual representations** of the objects it is passed as parameters, and slots them into the appropriate places within the **format string** it is passed. So this line of code is basically equivalent to:

  Console.WriteLine("Number " + day.ToString("d").ToString() + " of EnumTest.DAYS is " + day.ToString());
- The ToString method can take a single IFormatProvider parameter which indicates how the string conversion should be conducted. Values for this parameter can include things like g, d, **x**, **f**, etc.

Operators

- C# has a number of standard operators, taken from C, C++ and Java. Most of these should be quite familiar to programmers.
- To overload an operator in a class, one defines a method using the **operator** keyword. For instance, the following code overloads the equality operator.

```
public static bool operator == (Value a, Value b) {
    return a.Int == b.Int
}
```

Where an operator is one of a logical pair, both operators should be overwritten if any one is.

Jump and Selection Statements

- The **break** statement breaks out of the **while** and **for** loops.
- The **continue** statement can be placed in any loop structure.
- The **goto** statement is used to make a jump to a particular labeled part of the program code.
- **If-else** statements are used to run blocks of code conditionally upon a boolean expression evaluating to true.
- **Switch** statements provide a clean way of writing multiple if - else statements.

Loop Statements

- **while loops**

  while (expression) statement[s]

- **do-while loops**

  do statement[s] while (expression)

- **for loops**

  for (statement1; expression; statement2) statement[s]3

- **foreach loops**

  foreach (variable1 in variable2) statement[s]

```
int[] a = new int[]{1,2,3};
foreach (int b in a)
   System.Console.WriteLine(b);
```

Classes and Structs

- Classes provide **templates** from which objects – instances of those classes, can be generated. A class specifies a type and the constitutive elements (type members) of that type.
- A class can specify two main kinds of type members:
  - A class can specify other **types** – both value and reference. Types can contain other types, that is known as **containment**, or else **aggregation**.
  - A class can specify **methods** – functions designed for reading and manipulating the value and reference types an instance contains.
- C# classes can inherit from a single base class or from any number of interfaces.

Namespaces

- **Namespaces** can be thought of as collections of classes; they provide unique identifiers for types by placing them in an hierarchical structure.
- To use the System.Security.Cryptography.AsymmetricAlgorithm class, specify it in the following statement:

  using System.Security.Cryptography;
- An alias for the namespace can be specified as using myAlias = System.Security.Cryptography;
- For instance, the following code states that the class Adder is in the namespace fred.math.

```
namespace fred {
   namespace math {
     public class Adder { // insert code here }
   }
}
```

Class Declaration

- Class declarations can have up to four different parts, surrounding the **class** keyword:

  attributes class-modifiers class class-base class-body

- The class-body element specifies type members. The following is an example of a class declaration:

```
public class Shape {
    // class-body
}
```


- **Attributes** can be posted at the front of a class declaration. These comprise user-defined meta-data about the class; information which can be brought out at runtime.

Class Declaration

- There are **seven** different - optional - **class modifiers**. Four of these – **public**, **internal**, **protected**, and **private** – are used to specify the access levels of the types defined by the classes.
  - The **public** keyword identifies a type as fully accessible to all other types.
  - If a class is declared as **internal**, the type it defines is accessible only to types within the same assembly (a self-contained 'unit of packaging' containing code, metadata etc.).
  - If a class is declared as **protected**, its type is accessible by a containing type and any type that inherits from this containing type.
  - Where a class is declared as **private**, access to the type it defines is limited to a containing type only.

Class Declaration

- The permissions allowed by **protected internal** are those allowed by the **protected** level plus those allowed by the **internal** level.
- The **new** keyword can be used for nested classes.
- A class declared as **abstract** cannot itself be instanced - it is designed only to be a base class for inheritance.
- A class declared as **sealed** cannot be inherited from.

Class Declaration

- The **class base** part of the class declaration specifies the name of the class and any classes that it inherits from.
- The following line declares a public class called **DrawingRectangle** which inherits from the base class Rectangle and the interface **Drawing**:

```
public class DrawingRectangle : Rectangle, Drawing
```

- **Interfaces** are declared in much the same way as standard classes, except that they use the keyword interface in place of the keyword class. For instance:

```
public interface Drawing
```

Methods

- Methods are operations associated with types.

  int sum = Arithmetic.addTwoIntegers(4, 7);

- A method declaration, specified within a class declaration, comprises a **method-head** and a **method-body**.
- The method-head is made up of the following elements (square brackets enclose those which are optional).

  [attributes] [method-modifiers] return-type method-name ([formal-parameter-list])

- Method attributes work in a similar way to those for classes.

Methods

- There are **ten method modifiers** that can be used. Four of these are the access modifiers that can be used in class declarations. These four work analogously to the way they work in class declarations. The others are the following:
  - **Abstract**: A method without specifying its body. Such methods are themselves termed abstract. A class contains an abstract method it cannot be instantiated.
  - The **static** modifier declares a method to be a class method (a method that can be invoked without an instance).
- **Namespaces** (Packages in Java) can be thought of as collections of classes; they provide unique identifiers for types by placing them in an hierarchical structure.

Polymorphism (Inherited Methods)

- C# supports two different ways of method overwriting - **hiding** or **overriding**. Note that the term 'overwrite' is a term we have devised to cover both hiding and overriding.
- Method overwriting makes use of the following three method-head keywords:

  new, virtual, override
- The main difference between hiding and overriding relates to the choice of which method to call where the declared class of a variable is different to the run-time class of the object it references.

Constants, Fields, Properties and Indexers

- **Fields** are variables associated with either classes or instances of classes.
- There are seven modifiers which can be used in their declarations. These include the four access modifiers **public**, **protected**, **internal** and **private** and the new keyword.
- The two remaining modifiers are:
  - **Static**: By default, fields are associated with class instances. Use of the static keyword, however, associates a field with a class itself, so there will only ever be one such field per class, regardless of the number of the class instances.
  - **Readonly**: Where a field is readonly, its value can be set only once.

Constants, Fields, Properties and Indexers

- **Constants** are unchanging types, associated with classes, that are accessible at compile time. Because of this latter fact, constants can only be value types rather than reference types.

  ```
  public const int area = 4;
  ```

- **Properties** can be thought of as **virtual fields**. From the outside, a class' property looks just like a field. But from the inside, the property is generated using the actual class fields.
- If properties are virtual fields, **indexers** are more like **virtual arrays**. They allow a class to emulate an array, where the elements of this array are actually dynamically generated by function calls.

Classes, Objects, and Methods

```
class StackClass
{
  string myString;
  private int [] stack_ref;
  private int max_len, top_index;
  public StackClass() { // A constructor
    stack_ref = new int [100];
    max_len = 99;
    top_index = -1;
  }
  public void push (int number) {
    if (top_index == max_len) Console.WriteLine("Error in push-stack is full");
    else stack_ref[++top_index] = number;
  }
  public void pop () {
    if (top_index == -1) Console.WriteLine("Error in push-stack is empty");
    else --top_index;
  }
  public int top () {return (stack_ref[top_index]);}
  public bool empty () {return (top_index == -1);}
}
```

Classes, Objects, and Methods

```
class StockExample
{
  public static void Main (string[] args) {
    StackClass myStack = new StackClass();
    myStack.push(42);
    myStack.push(29);
    Console.WriteLine("29 is: {0}", myStack.top());
    myStack.pop();
    Console.WriteLine("42 is: {0}", myStack.top());
    myStack.pop();
    myStack.pop(); // Produces an error message
  }
}
```

Delegates and Events

- **Delegates** are reference types which allow indirect calls to methods.
  - A **delegate instance** holds **references to some number of methods**, and by invoking the delegate one causes all of these methods to be called.
  - The usefulness of delegates lies in the fact that the functions which invoke them are blind to the underlying methods.
- It can be seen that delegates are functionally rather similar to C++'s **function pointers**. However, it is important to bear in mind two main differences.
  - Firstly, **delegates are reference types** rather than value types.
  - Secondly, some **single delegates can reference multiple methods**.

Delegates and Events

- Delegates can be specified on their own in a namespace, or else can be specified within another class. In each case, the declaration specifies a new class, which inherits from System.MulticastDelegate.
- Each delegate is limited to referencing methods of a particular kind only.
  - The **type** is indicated by the delegate declaration – the input parameters and return type given in the delegate declaration must be shared by the methods its delegate instances reference.
- To illustrate this: a delegate specified as below can be used to refer only to methods which have a single String input and no return value.

Delegates and Events

- public delegate void Print (String s);
- Suppose, for instance, that a class contains the following method:

```
public void realMethod (String myString) {
    // method code
}
```

- Another method in this class could then instantiate the Print delegate in the following way, so that it holds a reference to realMethod;

```
Print delegateVariable = new Print(realMethod);
```

Delegates and Events

- We can note two important points about this example. Firstly, the unqualified method passed to the delegate constructor is implicitly recognized as a method of the instance passing it. That is, the code is equivalent to:

```
Print delegateVariable = new Print(this.realMethod);
```

- We can, however, in the same way pass to the delegate constructor the methods of other class instances, or even static class methods. In the case of the former, the instance must exist at the time the method reference is passed. In the case of the latter (exemplified below), the class need never be instantiated.

```
Print delegateVariable = new
Print(ExampleClass.exampleMethod);
```

Delegates and Events

- The second thing to note about the example is that all delegates can be constructed in this fashion, to create a delegate instance which refers to a single method.
- However, as we noted before, some delegates – termed **multicast delegates** – can simultaneously reference **multiple methods**. These delegates must - like our Print delegate – specify a void return type.
- The **method invocation** is termed an **event**, and the running of the method is the handling of the event. An typical example of an event is a user's selection of a button on a graphical user interface; this action may trigger a number of methods to handle it.
- The **event** keyword is used to **declare a particular multicast delegate**.

Delegates

```
using System;
using System.Threading;

public class Ticker
{
  private int i = 0;
  public void Tick(Object obj)
  {
    Console.WriteLine("tick " + ++i);
  }
}
public class DelegateExample
{
  static void Main(string[] args)
  {
    Ticker ticker = new Ticker();
    TimerCallback tickDelegate = new TimerCallback(ticker.Tick);
    new Timer(tickDelegate, null, 0, 500);

    Thread.Sleep(5000); // The main thread will now sleep for 5 seconds:
  }
}
```

Input/Output

```
using System;
using System.IO;

class DisplayFile
{
  static void Main(string[] args)
  {
    StreamReader r = new StreamReader(args[0]);
    string line;

    Console.Write("Out File Name: ");
    StreamWriter w = new StreamWriter(Console.ReadLine());

    while((line = r.ReadLine()) != null) {
      Console.WriteLine(line);
      w.WriteLine(line);
    }
    r.Close();
    w.Close();
  }
}
```

Exceptions

- The exception handling in C#, and Java is quite similar. However, C# follows C++ in allowing the author to ignore more of the exceptions that might be thrown (an exception which is thrown but not caught will halt the program and may throw up a dialogue box).
- To catch a particular type of exception in a piece of code, you have to first wrap it in a **try** block and then specify a **catch** block matching that type of exception.
- When an exception occurs in code within the try block, the code execution moves to the end of the try box and looks for an appropriate exception handler.

Exceptions

- For instance, the following piece of code demonstrates catching an exception specifically generated by division by zero:

```
try {
  res = (num / 0);
  catch (System.DivideByZeroException e) {
     Console.WriteLine("Error: an attempt to divide by zero");
  }
}
```


- You can specify multiple catch blocks (following each other), to catch different types of exception. A program can throw exceptions - including customized exceptions.

Exceptions

```
using System;
public class ExceptionDemo {
   public static void Main () {
    try {
      getException();
    } catch (Exception e) {
      Console.WriteLine("We got an exception");
    }
    finally {
      Console.WriteLine("The end of the program");
    }
  }
  public static void getException() {
    throw new Exception();
  }
}
```

Using the C# Compiler

- As we have noted earlier, C# classes are compiled in the first place to the Common Language Runtime Intermediate Language (IL).

  csc file.cs (mcs file.cs in mono)

- Where the required classes are held in more than one file, these should be listed, separated by spaces, as in:

  csc file1.cs file2.cs

- Broadly speaking, one can compile C# classes into either executable files or dynamic link library - DLL - files with the /t switch.
- Preprocessor directives tags included within class specifications; they are used to give the compiler additional information about regions of code.

Code Documentation

- The C# compiler supports the automatic creation of class documentation.
  - Where the equivalent functionality for Java produces HTML, the C# documenter produces XML.
  - This means that the C# documentation is not as immediately ready to use as the Java documentation.
  - However, it does allow there to be different applications which can import and use the C# documentation in different ways. (Note: using Visual Studio you can also create HTML documentation, but we will not be covering this here).
- To document any element in a C# script, you precede the element with XML elements. Each of the lines comprising this documentary code should be marked off as comments using the following special comment indicator (you can compare this with the standard comment indicators).

Code Documentation

- The following code gives an example of how one can provide overview information about a class.

```
/// <summary>
/// The myClass class represents an arbitrary class
/// </summary>
public class myClass
```

- Generating C# Documentation
  - You tell the compiler to produce documentation when compiling by invoking it with the switch: /doc:file
  - In this switch, file represents the name of the file that you want the documentation written to. As the documentation is generated in xml format, the file should have the extension .xml. So, for instance, to produce the documentation for a program in sys.cs file in a file named my.xml, we would use the command: csc sys.cs /doc:my.xml

Socket Programming (TCP)

- The C# API provides TCP streams by the following classes:
  - **TcpListener** - This class listens for connections from TCP network clients.
  - **TcpClient** - This class Provides client connections for TCP network services.
  - **Socket**- implements the Berkeley sockets interface.
- TcpListener(IPAddress, Port):
  - **AcceptTcpClient** - a blocking method that returns a TcpClient you can use to send and receive data. **Start** - Starts listening for incoming connection requests.
  - **AcceptSocket** - a blocking method that returns a **Socket** you can use to send and receive data.
  - **Close** - Closes the listener.

Socket Programming (TCP)

- **TcpClient (string *hostname*, int *port*);** :
  - **GetStream -** returns the **NetworkStream** used to send and receive data.
  - **Close**: Closes the TCP connection and releases all resources associated with the TcpClient.
- **NetworkStream**
  - **int Read** (byte[] *buffer*, int *offset*, int *size)* - Reads data from the NetworkStream.
  - **Write** (byte[] *buffer*, int *offset*, int *size)* - Writes data to the NetworkStream.
  - **Close** - Closes the NetworkStream.

Socket Programming (UDP)

- The C# API provides UDP streams by the following classes:
  - **UdpClient** - This class Provides User Datagram Protocol (UDP) network services.
- UdpClient Constructor:
  - **public UdpClient(** AddressFamily *family* **);**
  - **public UdpClient(** int *port* **);**
  - **public UdpClient(** IPEndPoint *localEP* **);**
  - **public UdpClient(** int *port*, AddressFamily *family* **);**
  - **public UdpClient(** string *hostname*, int *port* **);**
- IPEndPoint: Initializes a new instance of the IPEndPoint class.
  - **public IPEndPoint(** long *address*, int *port* **)**
  - **public IPEndPoint(** IPAddress *address*, int *port* **);**

Socket Programming (UDP)

- **UdpClient**:
  - public byte[] **Receive**(ref IPEndPoint remoteEP): Returns a UDP datagram that was sent by a remote host.
  - **Send**: Sends a UDP datagram to a remote host.
    - public int Send(byte[] dgram, int bytes)
    - public int Send(byte[] dgram, int bytes, IPEndPoint endPoint);
    - public int Send(byte[] dgram, int bytes, string hostname, int port);
  - Closes the UDP connection.