C# LANGUAGE: (by Buttolo Marco)

# INTRODUCTION

The C # language is essentially a language created by Microsoft, and is derived from C and C + +. It 'an object oriented language, which certainly presents many features that were present in C and C + +, but it also eliminates other such as macros, templates, and multiple inheritance. Have been added, however, that the functionality in C and C + + were not present as for example a greater safety in the types, version management, and so on. The C # language is a compiled language. The C # language is the language of the tip of the platform. NET Microsoft. This platform is composed of the main compilers for languages ​​supported by Microsoft, the CLR, and a set of large libraries. First, the **CLR (Common Language Runtime)** is the execution engine for. NET platform, and can be compared to the Java Virtual Machine for the Java language. Compilers in the frame work. NET compile the source code generating **language IL (Intermediate Language)** that runs right by the CLR. Let's get straight into the hours of the course of C #. We analyze the following code snippet:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Primo programma in c#");

}

}

}

As you can easily see, C # is an object-oriented language completely full, and all source code is within a class, just like in Java. The method of the main class (the entry point of the program) is the **main**. Education Console.WriteLine () allows you to write a message on the screen. This method resides in the System class imported through the Command **using**. Even with C #, like Java, C + +, C, each statement must be terminated by a semicolon. The following code snippet shows the declaration of a variable of type whole, and the display of its value:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int indice;

indice=2;

Console.WriteLine(indice);

}

}

}

The following code fragment illustrates how to declare a variable instead of type string:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

string testo;

testo="first example!!";

Console.WriteLine(testo);

}

}

}

So far we have seen how to declare integer types and string types. Let us now discuss in more detail the other items have not yet been clarified in the above code fragment. First, let's start with the fact that the concept of class is well defined and known. A class is a container of variables and methods to process these variables. The following code snippet shows the variables of interest have names and are of type string. The method setta\_variabili (), assigns a value to these variables. The method is declared as **public** and therefore visible to all classes. To call this method, you must instantiate the class Program, namely to create an object of class Program, and use that object with the dot notation to call the desired method.

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

//definition variable type

string name;

string surname;

static void Main(string[] args)

{

Program p = new Program();

p.setta\_variabili();

}

public void setta\_variabili()

{

//set values

name = "Marco";

surname = "Buttolo";

}

}

}

# THE VARIABLES

Let's see what are the possible checks that can be performed on variables:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

//definizione tipi variabili

string name;

string surname;

static void Main(string[] args)

{

Program p = new Program();

p.setta\_variabili();

if (p.name== string.Empty)

Console.WriteLine("Name empty");

}

public void setta\_variabili()

{

//setta valori a variabili

name = "";

surname = "Buttolo";

}

}

}

Using **String.Empty** is checked whether the string is empty or not. Note the classic use of the RU. If a cycle (or if a) has within it multiple statements, it is necessary to use the braces. In all examples, the term appeared namespace. Let's see what it means.

The base variables type used in C# are the same of other languages (integer, double, char, string,....).

# THE NAMESPACE

First, the namespace is an innovative technique that allows a developer to combine several different classes. For example:

namespace ConsoleApplication1

{

class Program

{

}

Class Program2

{

}

}

As can be seen from this example, the namespace ConsoleApplication1, has built two different classes. The namespaces are always public, and may be nested one inside the other. For example:

namespace primo

{

namespace secondo

{

public class Classe1

{ }

public class Classe2

{ }

}

}

primo.secondo.Classe1 c1 = new primo.secondo.Classe1();

primo.secondo.Classe2 c2 = new primo.secondo.Classe2();

As shown in this example, you can instantiate a class by specifying the namespace to which it belongs and using the dot notation. When, at the beginning of each program you will find a statement such as: using System; actually we only include its namespace. Previously we mentioned to the use of the public keyword when declaring a particular method. In particular, there are three types of keywords to be able to declare a particular method:

1. **Public**
2. **Private**
3. **Protected**

Using the first is done so that the method is visible to all instances of the class, while with private is made so that the method is not visible from outside the class. By default the value is Private.

All the methods so far seen are traditional methods, so to speak, that is, methods that are declared but mostly used with the object and the classical dot notation. It happens sometimes that you want to invoke a method belonging to a certain class without wanting to instantiate the class. This thing can be done by simply declaring the method as static and recalling the same with the class name and the dot notation, as is shown below:

class test{

static prova()

{

}

}

Test.prova();

Now let's see some basic types of language constructs and C #. We follow, by way of example, the following code:

using System;

using System.Collections.Generic;

using System.Text;

namespace struttura

{

public class gestione

{

public struct Persona

{

public string Name;

public string Surname;

public string Address;

//costruttore della classe

public Persona(string Name, string Surname, string Address)

{

this.Name = Name;

this.Surname = Surname;

this.Address = Address;

}

}

//entry point del programma

public static void Main()

{

Persona p = new Persona("Marco", "Buttolo", "xxx");

System.Console.WriteLine("Nome:" + p.Name);

System.Console.WriteLine("Cognome:" + p.Surname);

System.Console.WriteLine("Indirizzo:" + p.Address);

}

}

}

As you can easily see, with the keyword **struct**, you can create a data type that incorporates, inside, variables and methods to process these variables. The structure is a data type borrowed from C and C + +. The manufacturer instead is a special method that is called when you make the instantiation of the class. This method initializes the object of the class with the properties own in the constructor. It 'also important to note that the this keyword allows the programmer to refer to the variables of the current class. At this point, since we talked about methods, and how they can be declared public, protected or private you must provide a brief description of how C # are actually passed as parameters to these methods.

# PARAMETER TYPES

In C # there are three possible ways of passing parameters to a method:

1. **In**
2. **Out**
3. **Ref**

Let us briefly see how they work. In the method is the classic pass parameters by value, and therefore the method is changed to the value of the variable passed, it loses that value once out of the way. This is, by default, the above passage. Out and Ref are similar, ie represent the passage of parameters by reference, and allow to keep the change on the variable turning point in the method. The big difference between the two is that ref requires the initialization of the variable before it passes out while no. Below shows three examples of calls to the methods just mentioned:

public void metodo(int N){ ..... }

public void metodo(out int N){ ... }

public void metodo(ref int N){ .... }

If no keyword is not written before the passage of the parameter value is for (In).  
  
Even in C #, since this is an object oriented language, it is possible to define the concept of class inheritance. In the frame work. NET all classes inherit from a super class called **Object**. So, in C #, you can define a class that inherits from the first two in this way:

public class classe1 : classe2

{

}

So using the notation with colons is possible to derive a class from another. In C # **inheritance is single and not multiple** such as C + + or Python. In order to circumvent this limitation you can use, as we shall see later, the interfaces. The three key words of object-oriented programming are:

1. Inheritance
2. Polymorphism
3. Instantiation

The first and the third concept have already been addressed, while the second does not yet. For **polymorphism** means essentially the ability to define methods with the same name. Consider, by way of example, to two classes. A class vehicle and a car subclass that inherits methods and variables of the class vehicle. In fact a car is a vehicle. Now, suppose you want to create a method that prints the license plate of a vehicle, and for convenience we call this method with the name: “stampa\_targa()”. It 'clear that such a method that is present both in the class vehicle is in class car. Indeed, an automobile which is a particular case of vehicle has a number plate. The big difference, however, lies in the fact that the method “stampa\_targa” present in class car can be different compared to the same method in its super class. In the main class we will use the virtual keyword to indicate that this method may have the same definition in a subclass.

  public virtual void Stampa\_targa()  
  {  
    …….

  }  
  //...  
}

In the subclass though you must use the keyword ovverride to indicate that this method overwrites the same method in the superclass. Another very important concept is that overloading the definition of a particular method. In fact, a method is composed of a name and a list of parameters.

Per esempio:

public static void Stampa(string Messaggio)

{

System.Console.WriteLine("Il messaggio è: " + Messaggio);

}

public static void Stampa(int Numero)

{

System.Console.WriteLine("Il numero è: " + Numero);

}

As you can easily notice only changes the setting between one method and another. This is a case of overloading. We have seen up to now various concepts of C #, various keywords, and so on. We have seen how one defines the inheritance in C #. Now we just have to explain the concept of **abstract class**. To abstract class refers to a class that is instantiated with the new keyword and therefore is used only if it is derived. To declare a class as abstract simply use the abstract keyword as shown below:

public abstract class Figura

{

}

An abstract class to be used must be inherited. It 'important to note that an abstract class is only to be a kind of generalization. Interface, as has been said before is a technology that allows you to bypass the problem single inheritance. In a nutshell, an **interface** is a set of abstract methods that are implemented appropriately within the code where you use interface containing these methods. To use an interface, you use the interface keyword. Let's see an example:

public interface Interfaccia

{

void metodo1();

void metodo2();

......

}

public class classe : Interfaccia

{

public void metodo1()

{

codice......

}

}

The name of each interface must begin with the letter I. All that is defined inside an interface must be public. In addition, an interface can not contain variables. Through the two points you specify which interface to use in the classroom. Let us now see a whole series of simple examples of use of the most common cycles in C #:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int index;

for (index = 0; index < 20; index++)

{

Console.WriteLine("\n"+index);

}

}

}

}

Note that the for loop is identical to the for loop in C + + and C. "\ n" allows the carriage return.

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int index = 0;

while (index < 10)

{

Console.WriteLine("xxx");

index = index + 1;

}

}

}

}

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int index=0;

do

{

Console.WriteLine("xxx");

index = index + 1;

} while (index < 10);

}

}

}

The last type of construct that we analyze is the switch-case construct. This construct substantially allows to manage a whole series of case studies. for example

1. CASE A -> display a menu

2. CASE B -> open a file

3. CASE C -> delete a file

4. …….

The following code shows the use of switch case:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int caseSwitch = 1;

switch (caseSwitch)

{

case 1:

Console.WriteLine("Case 1");

break;

case 2:

Console.WriteLine("Case 2");

break;

default:

Console.WriteLine("Default case");

break;

}

}

}

}

As can be easily noted such constructs are all similar to the constructs present in the usual programming languages ​​such as C, C + +, Java, and so on.

# 5 STREAM FOR THE FILES

Until now C # has been introduced through examples, in a very simplified because the concepts therein are exposed traditional concepts present in all object oriented programming languages​​. But now we open a little discussion on two very important concepts: file handling in C #, and event management. We begin the discussion with simultaneous files describing both the concept of stream is also introducing the concept of form-oriented programming events and then programming under Windows (Visual C #). Let us first define what a stream player. A stream is essentially a class that allows to provide a methodology for displaying a sequence of bytes. A stream can then be viewed as a data queue in which it is possible to perform three operations:

1. Inserting data

2. Data Extraction

3. Searching for Data

the **filestream** is a particular type of stream that allows the programmer to write / read from a file, or retrieve data from a file. The following code fragment illustrates the use of the filestream.

using System;

using System.Collections.Generic;

using System.Text;

using System.IO;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

FileStream stream = File.Open(@"C:prova.txt", FileMode.Open);

StreamReader reader = new StreamReader(stream);

Console.WriteLine(reader.ReadToEnd());

reader.Close();

}

}

}

The example is pretty trivial. Imports the System.IO class is created called **FileStream** stream, and is associated with the Stream Reader. Reads the file, and then closes. The code is pretty self-explanatory. The **FileMode** allows you to specify how the work on file, and the method **ReadToEnd** to read until the end of the file opened previously. The stream reader, associated with the main stream, allowing just read it. Similarly, you can write a file:

using System;

using System.Collections.Generic;

using System.Text;

using System.IO;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

FileStream stream = File.Open(@"C:prova.txt", FileMode.Open);

StreamWriter writer = new StreamWriter(stream);

writer.WriteLine("test");

writer.Close();

}

}

}

As you can easily see, using StreamWriter can write to the stream. E 'clearly important in certain situations, as for example in the step of opening a file check if the latter exists. In fact, if we we open any file for reading, and this file does not exist that would be raised, which in computer jargon, is called an **exception**. In essence, an exception is a particular error that should be handled by the application. For the previous example of opening a file, it could easily happen that the file is not in the desired folder. This is an example of the exception. Let's see how the exception is handled in C #. To handle an exception you simply use the construct

try {

//Codice che potrebbe sollevare una eccezione.

} catch {

//Codice da eseguire in caso di eccezione.

} finally {

//Codice da eseguire in ogni caso

}

As you can easily notice, the code that allows for the opening of a file must reside within the block delimited try .... catch, and the exception is handled after the catch. It 'worth noting that in the catch clause can also enter the type of exception to handle. For example, if you have to handle the exception due to the impossibility to divide a number by zero, then the special exception will:

catch (DivideByZeroException ex)

Now we will see a number of examples of using the newly introduced concepts in theory, describing any new concepts. Let's see this very first example:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int i = 123;

object o = (object)i; // boxing

Console.WriteLine(o);

}

}

}

In this very first example makes use of the concept of boxing. First, the **boxing** is essentially forced conversion from one type (for example an integer) to a generic type object. The opposite operation known with the term of **unboxing** allows to extract the value type from the type object, as shown in the following code fragment:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int i = 452;

object o = i;

int j = (int)o;

Console.WriteLine(j);

}

}

}

Why the boxing and the unboxing is explained immediately. Take, as an example, the Java programming language. In Java there are special classes called wrapper classes that allow you to "wrap" existing classes in other classes. This type of classes is used for example on the data of the primitive type. In fact, the primitive type “int” in Java is converted into integer thanks to these classes, so that it is in all respects an object and is manageable as such, having at its disposal a number of methods. In C #, is done the same thing using these two techniques mentioned above. Now let's see a simple example of how you can open a web page in C #. Consider the following example:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

// apre una pagina web

System.Diagnostics.Process.Start("http://www.google.it/");

}

}

}

In a similar way, if you want to create a small program in C # that allows us to simply open the Windows Notepad utility, just write the following code fragment:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

// apre applicazione notepad

System.Diagnostics.Process.Start("notepad.exe");

}

}

}

# THE STRINGS

Let us now analyze in a somewhat 'detailed how to work with strings in C #. First, for those working in the field of computer science, the concept of string is a basic concept and should be in mind at all. Briefly, a string is a concatenation of multiple characters. Examples of string are:

love

hello

Given a string is clearly possible to calculate its length. You can do this simply by using a method called length. See about the follow example:

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string prova="test";

System.Console.WriteLine(prova.Length);

}

}

}

Now suppose we want to perform a concatenation of two generic strings. For example, suppose you want to concatenate two strings str1 and str2 respectively called initialized as follows:

str1=”Marco”

str2=”Buttolo”

The result of the concatenation is: "MarcoButtolo"

Now we see the following example:

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string str1="Marco";

string str2 = "Buttolo";

System.Console.WriteLine(str1+str2);

}

}

}

Now we analyze the following example:

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string s1 = "Prova";

string s2 = "Prova";

string s3 = "prova";

int result1 = String.Compare(s1, s2);

int result2 = String.Compare(s1, s3);

int result3 = String.CompareOrdinal(s1, s2);

Console.WriteLine(result1);

Console.WriteLine(result2);

Console.WriteLine(result3);

}

}

}

In the latter example shows how to perform the comparison between two or more strings, using the method **compare**. This method has as input parameters to compare strings, and returns zero if the strings are equal, otherwise returns 1 if the two strings are different. Please pay attention to the fact that method very similar to **strcmp** in C. In fact, the C # language is said to c-like language. The method **CompareOrdinal** essentially allows a tighter control on strings by performing a control character by character. Instead, using the **IndexOf** method you can search for text in a string specifying both the position from which to search, and the number of characters that comprise the string to be searched.

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string s1 = "Testretest";

string s2 = "te";

int result1 = s1.IndexOf(s2);

int result2 = s1.LastIndexOf(s2);

Console.WriteLine(result1);

Console.WriteLine(result2);

}

}

}

Other simple and useful functions for lavorrare with strings in C # are:

• String.Trim, which allows you to remove whitespace at the beginning and at the end of the string;  
• String.End, which allows you to remove the white lettering at the end of the string;  
• String.Start, which allows you to remove the white characters at the beginning of the string;  
• String.Replace, which allows you to replace occurrences of a given string;

# DATABASE

Let's see how you can access a database in C #. Consider the DBMS MySQL5. Before addressing this issue it is necessary to provide a range of basic theoretical elements to understand what we're talking about. First, a **database** is a collection of tables that are related to each other in some way. A table is a structure of this type:

|  |  |
| --- | --- |
| FIELD1 | FIELD2 |
| Value | Value |
| Value | Value |

Then a table is composed of a header row containing the fundamental fields of the table itself, and a series of other rows call **records** that contain the values ​​for the various fields. As an example, consider a table called **anagrafica\_studenti**, it can have many different fields including name, last name, address, registration number, and so on.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Surname | Address | Code |
| Marco | Rossi | Viale Certosa | 123 |
| Laura | Bianchi | Via Manzoni | 321 |
| … | …. | ….. | …. |

A set of tables of this type linked by certain relationships takes the name of the relational database. A **DBMS** (DataBase Managament System) is a software system that allows you to manage and manipulate a relational database. In the market there are different types of DBMS is both free and paid (or unpaid). An example of DBSM fee is SQL Server from Microsoft, as an example of DBMS MySQL5 is free and can be downloaded from the Internet. Before showing an example of database access in C # is desirable to give some definition. A transaction is basically a sequence of operations that can obviously result in a success or a failure. A transaction, to be such, must enjoy the **ACID properties** (Atomicity, Consistency, Isolation, and **Durability** (Atomicity, Consistency, Isolation and Durability)). A typical use of the transaction is as follows:

* Execution of instruction “BEGIN TRANSACTION”
* Execution query
* In fault case, abort the transaction (rollback)
* Execution confirm transaction (Commit)

MySQL5, for example, supports various types of tables here are listed below:

1. ISAM is an old table model sequential access;

2. MyISAM for tables that occupy a larger size, and has a better indexing (default for MySQL);  
3. INNODB, which is an engine for storing data in MySQL, which supports ACID transactions;  
Now let's see how to access a table in Microsoft SQL Server DBMS. The following code snippet shows how to access a table stored in a database in SQL Server at Microsoft.

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using System.Data.SqlClient;

using System.Management;

namespace WindowsFormsApplication3

{

public partial class Form1 : Form

{

public const String TABELLA = "test1";

private System.Data.OleDb.OleDbDataReader oleDataReader;

public Form1()

{

InitializeComponent();

}

public void Form1\_Load(object sender, EventArgs e)

{

object[] obj = new object[100];

try

{

String str\_query\_ricerca = "SELECT \* FROM" + " " + TABELLA;

obj = connetti\_db(str\_query\_ricerca, "test.udl");

}

catch

{

MessageBox.Show("Attenzione: query di ricerca errata!!");

}

//scan vector

foreach (Object element in obj)

{

if (element != null)

MessageBox.Show(element.ToString());

else

continue;

}

}

private object[] connetti\_db(String stringa\_query, String nome\_udl)

{

System.Data.OleDb.OleDbConnection conn = new

System.Data.OleDb.OleDbConnection("File Name=" + nome\_udl);

string str\_result;

object[] obj=new object[1000];

Object[] result = new Object[100000];

Boolean read;

int j = 0;

try

{

//connection to db

string sql = stringa\_query;

//open connection

conn.Open();

//executa query

System.Data.OleDb.OleDbCommand command = new

System.Data.OleDb.OleDbCommand(sql, conn);

oleDataReader = command.ExecuteReader();

Object[] tmp = new Object[oleDataReader.FieldCount];

//cycle read records

if (oleDataReader.Read() == true)

{

do

{

int NumberOfColums = oleDataReader.GetValues(tmp);

//scan result vector.

for (int i = 0; i < NumberOfColums; i++)

{

result[j] = tmp[i]; //copy vector

j++;

}

read = oleDataReader.Read();

} while (read == true);

}

}

catch(Exception e)

{

MessageBox.Show("Attenzione: impossibile eseguire query!!");

}

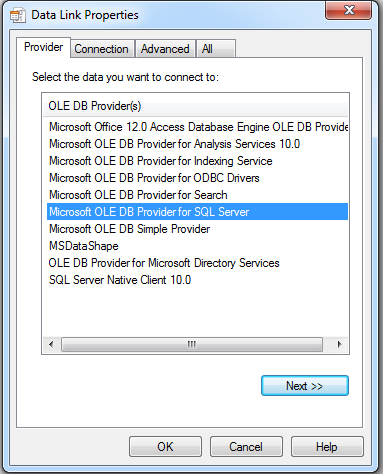
return result;

}

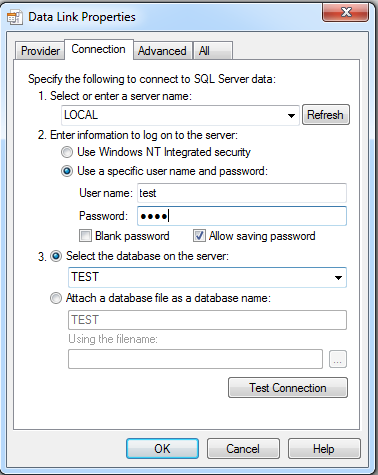
}

}

In this example the function is called “connetti\_db” passing the query string and the name of the UDL file from which to extract the parameters of the connection to the database in question. In few words, a UDL file is a file with a UDL (Universal Data Link) that stores the connection parameters to the database (eg: PC name / IP address that hosts the SQL server, the database name, the name of the table. ..). The following figure shows an example of UDL. As one can easily see, that file has various properties including the type of provider to which connect, the connection parameters, and so on.



Picture 1

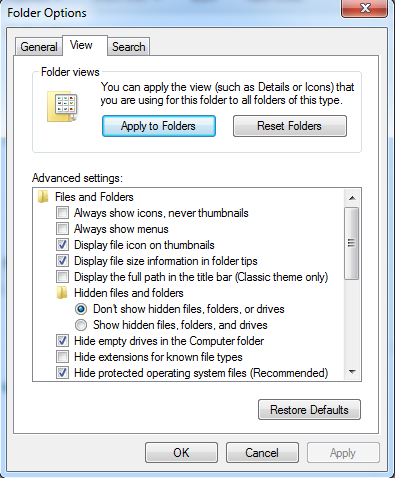


Picture 2

To create a file with a UDL there are at least two possible ways:

1. Click with the right mouse button on the desktop. Select New (New), then select the item Microsoft Data Link.

2. Click with the right mouse button and select New (New). Then select Text Document (text file). Change the extension TXT UDL (note: make sure that the option "Hide extensions for known file types" is disabled.)



Picture 3

With regard to the connection, using a connection-type OLEDB. In a nutshell, OLE DB (Object Linking and Embedding Database) is an **API (Application Program Interface)** from Microsoft for accessing data from a database. It 's the successor to ODBC. First you get a OLEDB connection to the desired database and then open the connection with the following statement:

//open connection

conn.Open();

You can then create an instance of **OleDbCommand** to execute commands. In our example, you want to take a reading and therefore the method **ExecuteReader** ().

This method makes it possible to send a command in text format on the connection channel and creates a OleDbDataReader, ie creates an instance of the **OleDbDataReader** which provides a stream unidirectional from the source of data to the application for the read operation of the same data . This class can not be inherited.

Note that the result of trivial query is stored in an array of generic type Object. In C #, all kinds of variables that are predefined or user-defined inherit directly or indirectly from Object. For variables of type object, you can assign values ​​of any type.