Visual Studio .Net

Visual Studio .NET ofera un set complet de instrumente de dezvoltare software, care permite realizarea, distribuirea şi rularea aplicaţiilor desktop Windows şi aplicaţiilor WEB. Tehnologia .NET utilizează mai multe limbaje de programare (VB, C++, C#, J#) și asigură atât portabilitatea codului compilat între diferite calculatoare, cât şi reutilizarea codului în programe, indiferent de limbajul de programare utilizat.
Un program scris într-un limbaj (oarecare) de programare acceptat de platforma .Net este supus unei compilări în urma căreia se obţine un cod scris într-un limbaj intermediar comun (CIL - Common Intermediate Language). În această formă aplicaţia este apoi trimisă procesorului virtual deb rulare CLR (Common Language Runtime) care realizează traducerea în cod maşină şi execută aplicaţia.
De remarcat că nivelul de abstractizare al codului in limbaj intermediar permite rularea aplicațiilor independent de platformă (cu condiția ca să existe o mașină virtuală pentru acea platformă).
In urma compilării unei aplicații rezultă un fișier cu extensia exe, dar care nu este direct executabil. Acest cod se va rula de către CLR.
CLR folosește tehnologia compilării JIT(Just in time compilation) - o implementare de mașină virtuală, în care o metodă sau o funcție, în momentul în care este apelată pentru prima oară, este tradusă în cod mașină. Codul translatat este depus într-un cache, evitându-se astfel recompilarea ulterioară. Există 3 tipuri de compilatoare JIT:
1. Normal JIT - a se vedea descrierea de mai sus.
2. Pre-JIT – compilează întregul cod în cod nativ o singură dată. În mod normal este folosit la instalări.
3. Econo-JIT - se folosește pe dispozitive cu resurse limitate. Compilează codul IL octet cu octet, eliberând resursele folosite de codul nativ ce este stocat în cache.

Visual Studio .Net utilizează două concepte:

- Proiect: fişier care conţine toate informaţiile necesare pentru a compila un modul dintr-o aplicaţie .Net
- Soluţie: fişier care conţine lista proiectelor care compun o aplicaţie, precum şi dependenţele dintre ele.

Proiectele sunt fişiere XML care conţin următoarele informaţii:

- lista fişierelor necesare, poziţia pe disc a acestora
- lista de module externe referite
- mai multe seturi de parametri de compilare
- diverse opţiuni legate de proiect

Fişierele de tip proiect pentru C# au extensia *csproj*.
Principalele tipuri de proiecte sunt:

- Console Application: aplicaţii de tip linie de comandă, fără interfaţă grafică; rezultatul este un fişier executabil (.exe)
- Class Library: bibliotecă de clase care poate fi utilizată pentru construirea altor aplicaţii; rezultatul este o bibliotecă cu legare dinamică (.dll)

- Windows Application: aplicaţie windows standard; rezultatul este un fişier executabil (.exe)
- ASP.NET WEB Application

Proiectele sunt singura modalitate prin care se pot compila aplicaţii .Net folosind VS.
Soluţiile sunt fişiere text cu extensia *sln* care conţin lista tuturor proiectelor care compun aplicaţia, dependenţele dintre ele şi configuraţiile disponibile.
Orice proiect este inclus obligatoriu într-o soluţie (creată explicit de către utilizator sau creată implicit de către VS).

Structura soluţiei poate fi vizionată folosind fereastra "Solution Explorer" (View->Solution Explorer).
Aplicaţia creată de VS poate fi rulată folosind CTRL+F5 (Debug->Start Without Debugging). În cazul în care o soluţie conţine mai multe proiecte, setarea proiectului care va porni la CTRL+F5 poate fi făcută prin right click pe proiect în "Solution Explorer" şi "Set as StartUp Project".
Proprietăţile proiectului pot fi accesate selectând proiectul în Solution Explorer + click dreapta Properties.

C#

Primul proiect : Conversii C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            double alfa=2.71, beta;
            string s1 = "3,14",s2;
            s2 = alfa.ToString();
            beta = double.Parse(s1);
            Console.WriteLine("beta={0}  s2={1}", beta, s2);
            Console.WriteLine("beta={0,1}  s2={1,10}",beta,s2);
            Console.WriteLine("beta={0,10:#.0000}  s2={1,-10}", beta, s2);

            Console.ReadKey();
        }



    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            string s;
            int varsta;
            string nume;
            Console.Write("nume=");
            nume=Console .ReadLine();
            Console.Write("varsta=");
            varsta=int.Parse(Console.ReadLine());
            s = new String('1',10); //s=”1111111111”
            Console.Write("studii=");
            s = Console.ReadLine();
            Console.WriteLine("{0} are {1} ani", nume, varsta);
            Console.WriteLine("{1} ani are {0}", nume, varsta);



            Console.WriteLine("studii: {0} ", s);

            Console.ReadLine();

        }
    }
}
```

Specificatorii de acces din C#

Controlul accesului la membrii unei clase se realizeaza prin utilizarea urmatorilor specificatori de acces:

-public (membrii publici pot fi accesati liber de codul din afara clasei);

-private (membrii privati sunt accesibili numai metodelor definite in aceeasi clasa);

-protected (membrii protejati pot fi accesati de metodele definite in cadrul aceleiasi clase sau de metodele definite in cadrul claselor care mostenesc clasa data);

-internal (specificatorul internal este utilizat pentru a declara membrii care sunt cunoscuti în toate fișierele dintr-un asamblaj, insa nu în afara asamblajului).

-protected internal (membrii sunt vizibili atat în clasele care moștenesc clasa în care sunt definiți acești membrii cât și în cadrul grupului de fișiere care formează asamblajul).

Cuvantul cheie static

In mod obișnuit, un membru al unei clase trebuie să fie accesat utilizând o instanță a acelei clase. Există însă posibilitatea ca un membru al unei clase să fie utilizat fară a depinde de vreo instanță. In acest caz, membrul respectiv este declarat ca static. Atat metodele cât și variabilele pot fi declarate statice. Variabilele declarate statice sunt variabile globale. La

declararea unei instanțe a clasei, variabilele statice nu sunt copiate. Ele sunt partajate de toate instanțele clasei. Variabilele statice sunt inițializate la încărcarea clasei în memorie. Dacă nu se specifică în mod explicit o valoare de inițializare atunci o variabilă statică se inițializează cu *0* dacă este tip numeric, cu *null* în cazul tipurilor de referința și respectiv *false* pentru tipul *bool*. Diferența dintre o metoda statică și o metoda obișnuita este că metoda statică poate fi apelată fiind prefixată de numele casei din care face parte, fără crearea unei instanțe a clasei.

Tipuri de parametri

In C# exista doua modalitati de transmitere al parametrilor către metode și anume: transferul prin valoare respectiv transferul prin referință. In mod implicit, tipurile valorice folosesc transferul prin valoare, in timp ce obiectele folosesc transferul prin referință.

Utilizarea modificatorilor *ref* și *out*
Corespunzător tipurilor valorice, în C# există trei tipuri de parametri:
Parametri de intrare: parametrii sunt implicit de intrare
Parametri de ieșire: sunt prefixați la definire și la apel de *out*
Parametri de intrare-ieșire: sunt prefixați la definire și la apel de *ref*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication3
{
    class Program
    {   static void suma(int a,int b, out int s)
         { s = a + b; }
        static void sch(ref int a, ref int b)
        {
            int c;
            c = a;
            a = b;
            b = c;
        }



        static void Main(string[] args)
        {
            int x = 10, y = 20,z;
            suma(x, y,out z);
            Console.WriteLine("{0}+{1}={2}", x, y, z);

            sch(ref x, ref y);
            Console.WriteLine("x={0}, y={1}", x, y);
            Console.ReadKey();
        }
    }
}
```

Spații de nume

Spațiile de nume sunt entităţi sintactice care permit gruparea logică a denumirilor de tipuri. Folosirea spațiilor de nume permite evitarea conflictelor generate de utilizarea aceloraşi identificatori în biblioteci diferite.
Declararea unui spațiu de nume se face folosind cuvântul cheie *namespace*:

```
namespace DenumireSpatiu
{
// declaratii
}
```

În cadrul *namespace-ului* tipurile sunt utilizate normal, iar în afara acestuia sunt utilizate folosind forma **`DenumireSpatiu.`***nume_tip*. Se pot declara *namespace-uri* imbricate pentru a construi o structură ierarhică de spații de nume. În cazul în care există mai multe declaraţii de spații cu acelaşi nume, ele sunt concatenate de către compilator.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace spatiul1
{
    class c1
    {
       public static int suma(int a, int b)
        {
            return a + b;
        }
    }
}
namespace spatiul2
{
    class c1
    {
      public static  int suma(int a, int b)
        {
            return a + b + a * b;
        }
    }
}
namespace SpatiiDeNume
{using spatiul1;
    class Program
    {
        static void Main(string[] args)
        {
           int x=c1.suma(5, 6);
           int y = spatiul2.c1.suma(5, 6);
```

```
            Console.WriteLine("x={0}  y={1}", x, y);
            Console.ReadKey();
        }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            string s;
            int varsta;
            string nume;
            Console.Write("nume=");
            nume=Console .ReadLine();
            Console.Write("varsta=");
            varsta=int.Parse(Console.ReadLine());
            s = new String('1',10);
            Console.Write("studii=");
            s = Console.ReadLine();
            Console.WriteLine("{0} are {1} ani", nume, varsta);
            Console.WriteLine("{1} ani are {0}", nume, varsta);



            Console.WriteLine("studii: {0} ", s);

            Console.ReadLine();

        }
    }
}
```

Accesori

Accesorii sunt metode care facilitează accesul la diferite atribute ale clasei. Deşi sunt utilizați la fel ca atributele, accesorii sunt de fapt metode şi nu reprezintă locaţii de memorie.

Declararea accesorilor se face sub forma:
tip NumeProprietate
{
  get { ... }
  set { ...}
}

După cum se poate observa, un accesor este alcătuit de fapt din două funcţii;
din declaraţia de mai sus compilatorul va genera automat două funcţii:
tip get_NumeProprietate() şi *void set_NumeProprietate(tip value)*.
Metodele de tip *set* primesc un parametru implicit denumit *value* care conţine valoarea atribuită proprietăţii.
Nu este obligatorie definirea ambelor metode de acces (get şi set); în cazul în care una dintre proprietăţi lipseşte, proprietatea va putea fi folosită numai pentru citire sau numai pentru scriere (în funcţie de metoda implementată).

Metoda ToString

Toate tipurile de date predefinite conţin metoda *ToString* (moştenită din *object* şi suprascrisă în clasele derivate) care permite transformarea valorii respective în *string*. În cazul tipurilor numerice, transformarea în *string* se poate face si cu formatare.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Unghiuri
{
    public class Unghi
    {
        int g, m, s;
        public Unghi(int g = 0, int m = 0, int s = 0)
        {
            this.g = g;
            this.m = m;
            this.s = s;
        }
        public Unghi(Unghi u)
        {
            g = u.g;
            m = u.m;
            s = u.s;
        }

        public int Grade
        {
            get { return g; }
```

```
            set { g = value; }
        }

        public override string ToString()
        {
            return "" + g + (char)176 + " " + m + "' " + s + "\" ";
        }
        public static Unghi operator +(Unghi u1, Unghi u2)
        {
            Unghi u = new Unghi();
            int t = 0;
            u.s = u1.s + u2.s;
            t = u.s / 60;
            u.s %= 60;
            u.m = u1.m + u2.m + t;
            t = u.m / 60;
            u.m %= 60;
            u.g = u1.g + u2.g + t;
            return u;



        }

    }
    class Program
    {
        static void Main(string[] args)
        {
            for (byte i = 65; i < 255; i++)
                Console.Write("{0}-->{1}; ", i, (char)i);
            Console.WriteLine();
            Unghi u1 = new Unghi(30, 40, 15);
            Unghi u2 = new Unghi(40, 30, 45);
            Unghi u = u1 + u2;

            Console.WriteLine("u=" + u.ToString());
            Console.ReadKey();


        }
    }
}
```

TRATAREA EXCEPTIILOR

Prin excepţie se înţelege un obiect care încapsulează informaţii despre situaţii anormale în funcţionarea unui program.

De exemplu: erori la deschiderea unor fişiere, împărţire la 0 etc. Aceste erori se pot manipula astfel încât programul să nu se termine abrupt. Sunt situaţii în care prevedem apariţia unei erori într-o secvenţă de prelucrare şi atunci integrăm secvenţa respectivă în blocul unei instrucţiuni **try**, precizând una sau mai multe secvenţe de program pentru tratarea excepţiilor apărute (blocuri **catch**) şi eventual o secvenţă comună care se execută după terminarea normală sau după ”recuperarea” programului din starea de excepţie (blocul **finally**).

In C# se pot arunca ca exceptii obiecte de tip System.Exception sau derivate ale acestuia. Exista o ierarhie de exceptii care se pot folosi, sau se pot crea propriile tipuri exceptie.

Aruncarea cu throw
Aruncarea unei exceptii se face folosind instructiunea throw. Exemplu:
throw new System.Exception();

Prinderea cu catch
Prinderea si tratarea exceptiei se poate face folosind un bloc catch, creat prin intermediul instructiunii catch.

Blocul finally
Uneori, aruncarea unei exceptii si golirea stivei pana la blocul de tratare a exceptiei poate sa nu fie o idee buna. De exemplu, daca exceptia apare atunci cand un fisier este deschis (si ınchiderea lui se poate face doar ın metoda curenta), atunci ar fi util ca sa se ınchida fisierul ınainte ca sa fie preluat controlul de catre metoda apelanta. Altfel spus, ar trebui sa existe o garantie ca un anumit cod se va executa, indiferent daca totul merge normal sau apare o exceptie. Acest lucru se face prin intermediul blocului finally, care se va executa ın orice situatie. Existenta acestui bloc elimina necesitatea existentei blocurilor catch (cu toate ca si acestea pot sa apara).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MI_C1
{
    class Program
    {
        public static  bool IsNumeric(object v)
        {
            try
            {
                double d = double.Parse(v.ToString());
                return true;
            }
            catch (Exception)
            { return false; }
        }

        static void Main(string[] args)
        {

            int d = 25;
            string sir;
            sir = "" + d;
            Console.WriteLine("sir={0}", sir);
            sir = d.ToString();
```

```
            Console.WriteLine("sir={0}", sir);

            sir = "234,567";
            double t;
            if(IsNumeric(sir))
             { t = double.Parse(sir);
              Console.WriteLine("t={0}", t);
             }

            else

            Console .WriteLine ("sir nenumeric");



            Console.ReadKey ();
        }
    }
}
```