3.3. Limbajul C#

Limbajul C# este un limbaj imperativ, obiect-orientat. Este asemănător din punct de vedere sintactic cu Java și C++, astfel trecerea de la unul dintre limbaje la celălalt nu este foarte abruptă.

Orice program scris în C# conține o serie de elemente, exemplu:

```
using System;
class HelloWorld
{
	public static void Main()
	{
		Console.WriteLine("Hello world!");
	}
}
```

Prima linie *using System;* este o directivă care specifică faptul că se vor folosi clasele care sunt incluse în spațiul de nume *System.* Spațiul de nume este o colecție de tipuri sau de alte spații de nume care pot fi folosite într-un program. În spațiul de nume *System* se află și spațiul de nume *Console*.

Un program trebuie să conțină și o metodă *Main* care returnează o valoare întreagă; această metodă se execută la pornirea programului, reprezintă punctul de intrare în aplicație, poate să preia parametrii din linie de comandă și trebuie să returneze o valoare. Modificarea programului pentru a accepta parametrii din linie de comandă este următoarea:

```
public static void Main(String[] args)
{
	for( int i = 0; i < args.Length; i++)
	{
		Console.Writeline(args[i]);
	}
}
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

C# este un limbaj de programare orientat pe obiecte(object oriented programming) respectând toate principiile: încapsularea, supraîncărcarea, moștenirea, polimorfism. Toate componentele limbajului sunt într-un fel sau altul, asociate noțiunii de clasă. Chiar şi tipurile primitive: byte, int sau bool sunt clase sigilate derivate din clasa ValueType din spaţiul de nume System. Pentru a evita unele tehnici de programare periculoase, limbajul oferă tipuri speciale cum ar fi: interfeţe şi delegări. Versiunii 2.0 a limbajului i s-a adăugat un nou tip: clasele generice

Sintaxa de declarere a unei clase este următoarea:

```
[atribut] [modificator] class [nume_clasă] [: clasă_de_bază]
{
    corp_clasă
}
```

Clasa de bază reprezintă clasa care este părintele clasei curente, clasa curentă moștenind toate calitățile acelei clase.

Atributele oferă o metodă puternică de a asocia informații declarative cu cod C#.

Clasele pot conține constante, câmpuri, metode, proprietăți, evenimente, indexatori, operatori, constructori, destructori, tipuri imbricate. Modificatorii de acces oferă un nivel de protecție care controlează gradul de acces la componentele clasei. Modificatorii sunt ilustrați în **Tabelul 3.2** împreună cu semnificațiile lor.

Accesor	Semnificație
public	Acces nelimitat
protected	Accesul este limitat la clasa conținătoare și la cele care derivă din ea
internal	Accesul este limitat la assembly-ul curent
protected internal	Acces limitat la assembly-ul curent sau la clasele derivate din ea
private	Acces limitat doar la clasa conținătoare, acesta este și modificatorul implicit

Tabelul 3.2. *Modificatorii de acces*

Capitolul 3 – Tehnologii folosite în realizarea aplicației

Exemplu în care se folosesc de modificatorii de acces:

```
public class A
{
	public string Nume;

	private int cantitate;
	public int Cantitatea
	{
		get
		{
			return cantitate;
		}
		set
		{
			cantitate = value;
		}
	}
	public A( int cantitate )
	{
		this.cantitate = cantitate;
	}
	public void AfișeazăTest(string text)
	{
		Console.WriteLine("Test " + text);
	}
	protected void AfișeazăTest2(string text)
	{
		Console.WriteLine("Test2 " + text);
	}
}
```

```
public class B : A
{
	public B(string nume)
	{
		this.Nume = nume;
	}
	public void SchimbăCantitate(int cantitate)
	{
		// this.cantitate = cantitate; // nu funcționează din cauza gradului de acces
		AfișeazăTest2(cantitate.ToString()); // funcționează
	}
	private class b
	{	// clasă în clasă sau clasa imbricată   }
}
class Test
{
	static void Main()
	{
		A obiectA = new A(20);
		B obiectB = new B("TestB");
		// B.b obiectBb = new B.b(); // nu funcționează datorită nivelului de acces
		A.AfișeazăTest("Test");
		// A.AfișeazăTest2("Test2"); // nu fucționează
		B.SchimbăCantitate(20); // funcționează
		Mesaj();
	}
	static void Mesaj()
	{
		Console.WriteLine("Bye");
	}
}
```

Capitolul 3 – Tehnologii folosite în realizarea aplicației

Exemplificăm următoarea ierarhie de clase

public *abstract* class Mașină { }

public class Dacia: Mașină { }

public sealed class Trabant: Mașină { }

Modificatorul *abstract* este folosit pentru a desemna faptul că nu se pot obține obiecte din clasa Mașină, practic este o clasă ce nu poate fi instanțiată ci numai derivată, iar modificatorul *sealed* a fost folosit pentru a desemna faptul că nu se mai pot obține clase derivate din clasa Trabant (de exemplu, subclasa TrabantNou).

Constructorul este o metodă ce se execută la instanțierea unei clase, ea poate primi diverşi parametrii, o clasa poate avea mai mulți constructori.

Tipurile de valori întregi sunt cele mai simple tipuri de valoare pentru computer pentru a stoca. Fiecare valoare va mapa pe un model special de biți. Singura problemă este că permite stocarea valorilor dintr-un anumit interval. Cu cât este mai mare valoarea pe care vrem să o reprezintăm cu atât numărul de biți trebuie să fie mai mare. C # oferă o varietate de tipuri de întregi, în funcție de intervalul de valori pe care în care se dorește să se stocheze.

sbyte	8 octeți	-128 până la 127
byte	8 octeți	0 până la 255
short	16 octeți	-32768 până la 32767
ushort	16 octeți	0 până la 65535
int	32 octeți	-2147483648 până la 2147483647
uint	32 octeți	0 până la 4294967295
long	64 octeți	-9223372036854775808 până la 9223372036854775807
ulong	64 octeți	0 până la 18446744073709551615
char	16 octeți	0 până la 65535

Tabelul 3.3. *Tipurile întregi de dată*

Capitolul 3 – Tehnologii folosite în realizarea aplicației

"Real" este un termen generic pentru numere care nu sunt numere întregi. Ele au un parte zecimală şi o parte fracţionară. În funcţie de valorile zecimalelor, plutește în jurul numărului, de aici numele de float(plutire). Tipul standard *float* are un interval între 1,5E-45 și 3,4E48 cu o precizie de doar 7 zecimale. Dacă dorim mai multă precizie putem folosi tipul *double* care are o precizie de 15 zecimale, iar intervalul este de la 5.0E-324 până la 1.7E308. Mai este un tip, *decimal*, care are cea mai mare precizie 28 de zecimale.

Tipurile literale sunt *char* și *string*, care pot reține orice șiruri de caractere. Iar tipul rămas nemenționat încă este tipul *boolean* care are două valori de adevărat și fals (true / false).

Colecțiile generice au apărut împreună cu clasele generice. Listele, notate List<T> sunt colecții generice care sunt utile pentru manipularea șirurilor de date de acelaşi tip. Exemplu:

```
List<int> intList = new List<int>();
intList.Add(30);
intList.Add(6);
intList.Add(35);
```

Dacă încercăm să introducem un alt tip de dată decât int atunci va rezulta o eroare, lista nu permite decât tipul int.

Tratarea excepțiilor este o măsură de siguranță, C# oferă posibilitatea de a face față oricărei situații excepționale sau neprevăzute, care apare în timpul execuției unui program. Pentru a manipula excepțiile, limbajul folosește try, catch și finally, acestea fiind cuvinte cheie pentru a testa reușita anumitor acțiuni, pentru a stopa eșecul (failure) și pentru a șterge memoria. Excepțiile pot fi generate de către Common Language Runtime (CLR), de către alte biblioteci, sau din codul aplicației folosind cuvântul cheie throw.

Exemplu:

```
// împărțirea a 0

int ImparteLa(int nr, int imp)
{
```

```
	return nr / imp;
}
```

Acest cod nu este sigur, dacă se trimite ca parametru orice număr și zero atunci el va genera o excepție, care dacă nu este capturată în nicio porțiune de cod atunci toată aplicația va eșua. Pentru a face față oricăror numere transmise acelei metode trebuie să introducem un bloc try - catch în metodă. Metoda rescrisă va arăta astfel:

```
int ImparteLa(int nr, int imp)
{
	try
		{
			int rezultat = nr / imp;
			return rezultat;
		}
	catch(Exception)
	{
		return 0;
	}
}
```

În acest caz, orice excepție provocată de împărțirea celor două numere va provoca returnarea rezultatului zero.

Excepțiile mai pot fi provocate și intenționat folosind *throw* putem arunca excepții de oriunde, chiar din blocul de *catch*. În exemplul de mai sus, dacă metoda *ImparteLa* se află într-un bloc try-catch în metoda care o apelează atunci putem face verificarea înainte și aruncăm o excepție daca *imp* este zero.

```
int ImparteLa(int nr, int imp)
{
	if(imp == 0)
```

```
        {
                throw new Exception("Împărțire la zero!");
        }
        return nr / imp;
}

void functieX()
{
        try
        {
                int rez = ImparteLa(5, 0);
        }
        catch(Exception ex)
        {
                Console.WriteLine(ex.Message);
        }
}
```

La apelul *funcțieX()* se intră în funcția *ImparteLa*, *imp* fiind egal cu zero se intră pe ramura if-ului și se aruncă o excepție cu mesajul: "Împărțire la zero!", această excepție este capturată în blocul de catch al funcției *funcțieX* și se afișează în consolă mesajul excepției.

Clasele care permit lucrul cu baza de date SQL se află în spațiul de nume *System.Data.SqlClient.* Pentru persistență avem nevoie de trei clase: *SqlConnection, SqlCommand* și *SqlDataReader.*

SqlConnection este folosit pentru a deschide o conexiune cu serverul SQL pentru a accesa baza de date. Primește ca parametru un string de conexiune care conține informații despre conexiune: se specifiă serverul, username-ul parola și baza de date cu care se va lucra. Stringul de conexiune arată astfel:

```
"Data Source=ADI-PC\SQLEXPRESS;Initial Catalog=GestiuneFirma;Persist Security
Info=True;User ID=sa;Password=1q2w3e4r;"
```

SqlCommand este folosită pentru a apela o procedură stocată deja existentă, prin metoda *AddWithValue* putem adăuga parametrii necesari pentru procedură:

```
SqlConnection con = new SqlConnection(connectionString); // specificare conexiunii
SqlCommand cmd = new SqlCommand(numeProceduraStocata, con);
cmd.CommandType = System.Data.CommandType.StoredProcedure;
con.open(); // se deschide conexiunea cu baza de date
SqlDataRead reader = cmd.ExecuteReader(); // execută procedura stocată din baza de date
while(reader.Read()) // reader face citirea fiecărui rând din interogarea rezultată
{
        obiect = reader[0],
        .......
        // transformarea datelor citite din baza de date în echivalentul lor din C#
        .......
}
reader.Close();
con.Close();
```

Reader-ul și conexiunea trebuie închise după ce s-a încheiat lucrul cu baza de date, pentru că altfel pot duce la rezultate neașteptat, chiar și excepții. Deschiderea unei conexiuni de mai multe ori duce la apariția unei excepții. Cea mai sigură metodă de a folosi conexiune este cu blocul *using*.

```
using(SqlConnection con = new SqlConnection(connectionString)
{
        // aici se face persistența
}
```

Folosirea blocului *using* provoacă distrugerea obiectului folosit la părăsirea blocului, asta presupune automat și închiderea conexiunii cu baza de date.