Saturs

Valoda C#

Valodas C# iespējas

Valoda C# tika izstrādāta speciāli NET videi. Tā ir pilnīgi objektorientēta valoda, kas apvieno sevī vislabākās C++, Java un Visual Basic valodu iespējas.

Salīdzinoši ar valodu C++, C# valodai ir vienkāršāka sintakse un nav dažu pārāk sarežģītu C++ valodas iespēju, piemēram: C# valodā nav rādītāju, C# valodā nav iespējams izveidot klasi, kas ir vairāku klašu pēctecis, bet šo C++ iespēju var imitēt izmantojot interfeisus.

Atšķirībā no C++ valodas, valodā C# visi mainīgie tiek inicializēti automātiski, un lai piekļūtu pie objekta metodēm operāciju :: un -> vietā var izmantot operāciju punkts (.).

Bez tam C# nodrošina vienotu tipu sistēmu, visi C# valodas tipi ir atvasināti no viena bāzes tipa System.Object. Tas dod iespēju apstrādāt visus tipus kā objektus, bet valodā C# ir saglabāta arī iespēja apstrādāt veselas vērtības kā parastas 32-bitu vērtības.

Programmā, pierakstītā valodā C#, var iestarpināt eksistējošu valodas C/C++ kodu, izmantojot speciāli apzīmētus blokus, kurus sauc par "nedrošā koda" (unsafe code) blokiem. Lai apzīmētu šos blokus tiek lietots vārds **unsafe**, un nedrošā koda blokos atļauts izmantot rādītājus, izpildīt aritmētiskas darbības ar tiem.

Valoda C# ir ļoti līdzīga valodai Java, bet ir arī daudz atšķirību. Piemēram, C# ir pilnība integrēta ar COM standartu, tas nozīmē, ka C# programmās var izmantot COM komponentus, neatkarīgi no tā, kādā valodā tie tika izstrādāti. IL kods, atšķirība no Java bait-koda tiek kompilēts, nevis interpretēts, tāpēc lietojumprogrammas, kas ir pierakstītas C# valodā, strādā parasti ātrāk. Dalītu (Enterprise tipa) programmu izstrāde valodā C# ir lētāka, nekā izstrāde valodā Java, tāpēc kā Enterprise līmeņa dienesti, vajadzīgi C# programmai, jau ir iekļauti Windows vidē, bet analoģiski Java risinājumi prasa ieinstalēt J2EE (Java 2 Enterprise Edition) dienestus, kuru cena ir augsta.

Vienkāršākā programma valodā C#

Vienkāršākā programma valodā C# izskatās sekojošā veidā:

```
using System;
public class Hello
{
 public static void Main(string [] args)
      {
       Console.WriteLine("Hello World!");
       Console.WriteLine("There were {0} arguments", args.Length);
      }
}
```

Vārds **using** tiek lietots, lai atsauktos uz klašu bibliotēkām, kas ir iebūvētas .NET Framework vidē. Klašu bibliotēka sastāv no vārdu telpām, kas satur hierarhiski organizētas klases. Dotā piemērā direktīva **using** dod iespēju piekļūt pie vārdu telpas **System**.

Izskatāma programma satur vienas klases **Hello** aprakstu. Klase **Hello** satur tikai vienu statisku metodi **Main(),** no kuras sākas programmas izpildīšana. Katrā C# programmā ir jābūt metodei ar vārdu **Main**. Šo metodi jānodefinē klases vai struktūras iekšā. Metodi **Main** ir obligāti jāapraksta ar pieejas modifikatoriem **public** un **static**. Metode **Main** var dod atpakaļ **void** vai **int** tipa vērtību. Metode **Main** var aprakstīt vai vispār bez parametriem, vai ar vienu parametru, kas ir pēc savas būtības rindu masīvs, kas satur komandrindas parametrus. Piemēram:

```
public static void Main()
public static int Main(string [] args)
public static void Main(string [] args)
```

Izskatāmā piemērā metode **Main** satur divus operatorus **Console.WriteLine**, kas izvada paziņojumu ekrānā.

Komentāri

Komentāra sākumu ir jāatzīmē ar simboliem /*, bet komentāra beigās ar */. Jebkuras simbolu virknes, kas atrodas aiz simboliem // līdz rindas beigām arī tiek uzskatītas par komentāru.

Konsoles ievades-izvades operācijas

Vārdu telpa **System** satur klases **Console** aprakstu. Dotā klase satur metodes, paredzētas datu ievadei un izvadei.
Izvadīt datus var ar metodēm **WriteLine** un **Write**. Metode **WriteLine** atšķiras no metodes **Write** ar to, kā pārvieto kursoru jaunajā rindā. Piemēram:

```
string str1 = "Console";
string str2 = "System";
Console.WriteLine("{0} is a class in the {1} namespace", str1, str2);
      // šeit {0} norāda vietu simbolu virknē, kura ir jāiestarpina
      // funkcijas WriteLine otru parametru (str1)
      // bet {1} – norāda, kur iestarpināt trešu parametru (str2)
int i = 5;
Console.WriteLine(i);
```

Ievadīt simbolu virkni no tastatūras var ar metodi **ReadLine**. Piemēram:

```
string s = Console.ReadLine();
Console.WriteLine("Hello, {0}", s);
```

Nolasīt vienu simbolu un noteikt tā ASCII kodu var ar metodi **Read**. Piemēram:

```
int i = Console.Read();
Console.WriteLine(i);
```

Mainīgie

Visus mainīgos, kurus var izmantot, valodā C# var sadalīt septiņās kategorijās:

1) Statiskie mainīgie – tie ir lauki, kas tiek aprakstīti ar modifikatoru **static** un kas eksistē visā programmas izpildes laikā. Piemēram:

```
static int x=4;
```

2) Eksemplāra mainīgie – laiki, kas aprakstīti klases vai struktūras robežās bez modifikatora **static**. Eksistē tik ilgi, cik ilgi eksistē atbilstošais objekts vai struktūra. Piemēram:

```
decimal d = 2.1414;
```

3) Masīva elementi.

4) Formālie parametri, kas tika nodoti pēc vērtības – tas ir funkciju parametri, kas ir aprakstīti bez modifikatoriem **out** un **ref**. Tāds parametrs tiek veidots funkcijas izsaukšanas brīdī un tiek iznīcināts, kad funkcija beidz savu darbu. Izmaiņas, izdarītas ar tādu parametru funkcijas ķermenī, neietekmē uz sākotnējiem datiem. Piemēram:

```
using System;
public class Val
{
      static void AddTwo(int a)
      {
            a = a+2;
            Console.WriteLine("a = {0}", a);
      }
      public static void Main()
      {
            Console.WriteLine("Input number:");
            string str = Console.ReadLine();
            int i = Int32.Parse(str);
            Console.WriteLine("i = {0}", i);
            AddTwo(i);
            Console.WriteLine("i = {0}", i);
                  // izvadīs mainiga i sākumvērtību
      }
}
```

5) Formālie parametri, kas tiek nodoti pēc norādes – šie parametri tiek aprakstīti ar modifikatoru **ref**. Modifikatoru **ref** ir obligāti jānodrukā metodes virsrakstā un metodes izsaukšanas operatorā. Funkcija var izmainīt to mainīgo vērtības, kas tika nodotas tai pēc norādes. Piemēram:

```
using System;
class Reftest
{
      static void Change(ref string x, ref string y)
      {
            string temp = x; x = y; y = temp;
      }
      public static void Main()
      {
            string s = "abc";
            string t = "123";
            Change(ref s, ref t);
            Console.WriteLine("s = {0}, t = {1}", s, t);
```

```
                        // izvadīs s = 123, t = abc
        }
}
```

6) Atgriežamās vērtības – tās ir parametri, aprakstīti ar modifikatoru **out**. Parametri, kas ir aprakstīti ar modifikatoru **out,** ir līdzīgi parametriem, aprakstītiem ar modifikatoru **ref**, bet ja tiek lietots vārds **out**, tad ir atļauts neinicializēt atbilstošus mainīgos. Piemēram:

```
using System;
class Outtest
{
      static void Add(int x, int y, out int sum)
      {
            sum = x+y;
      }
      static void Main()
      {
            Console.WriteLine("Input first number: ");
            string s = Console.ReadLine();
            int i = Int32.Parse(s);
            Console.WriteLine("input second number: ");
            string t = Console.ReadLine();
            int j = Int32.Parse(t);
            int k;
            Add(i, j, out k);
            Console.WriteLine("{0}+{1}={2}", i, j, k);
      }
}
```

7) Lokālie mainīgie – tie ir mainīgie, kas tiek aprakstīti kādas metodes iekšā. Pie šiem mainīgiem var piekļūt tikai atbilstošā metodē.

Konstantes

Konstantēm ar modifikatora **const** palīdzību var piešķirt vērtības. Piemēram:

```
const int a = 50;
```

Dotajā piemērā konstantes **a** vērtību aizliegts mainīt.

Mainīgu tipi

Visus tipus var sadalīt trjās grupās:

1) vērtību tipi;
2) norādes tipi;
3) rādītāju tipi

Vērtību tipi

Mainīgie vērtības tipa (value type) glabā reālus datus. Kad tādi mainīgie tiek nodoti kādai funkcijai, tad tiek izveidotas šo mainīgo kopijas. Tas garantē to, kā viena funkcija nevarēs sabojāt datus, kurus izmanto cita funkcija. Vērtību tipus arī var sadalīt trjās grupās:

1) vienkāršie tipi;
2) struktūras;
3) uzskaitījumi.

Vienkāršie tipi

Sekojošā tabula satur vienkāršo tipu saīsinātus nosaukumus un līdzvērtīgus tiem sistēmas tipu vārdus. Saīsināti tipu vārdi var atšķirties dažādās .NET valodās, bet sistēmas tipu vārdi (pilnie tipu vārdi) visās valodās sakrīt.

Tips	Sistēmas tips	Paskaidrojums
`byte`	`System.Byte`	8-bitu skaitlis bez zīmes
`sbyte`	`System.Sbyte`	8-bitu skaitlis ar zīmi
`short`	`System.Int16`	vesels ar zīmi (aizņem 2 baitus)
`ushort`	`System.UInt16`	vesels bez zīmes (aizņem 2 baitus)
`int`	`System.Int32`	vesels ar zīmes (aizņem 4 baitus)
`uint`	`System.UInt32`	vesels bez zīmes (aizņem 4 baitus)
`long`	`System.Int64`	vesels ar zīmes (aizņem 8 baitus)
`ulong`	`System.UInt64`	vesels bez zīmes (aizņem 8 baitus)
`char`	`System.Char`	Unicode simbols (2 baiti)
`float`	`System.Single`	reālais skaitlis (4 baiti)
`double`	`System.Double`	reālais skaitlis (8 baiti)
`bool`	`System.Boolean`	loģiskais tips (true, false)
`decimal`	`System.Decimal`	decimālais skaitlis ar 28 zīmīgiem cipariem

Struktūras

Struktūras ir līdzīgas klasēm un var saturēt laukus, īpašības, metodes, konstruktorus. Struktūras atšķiras no klasēm ar to, ka nenodrošina mantošanu, bet līdzīgi klasēm, struktūras var realizēt interfeisus. Otra atšķirība ir saistīta ar to, ka struktūras ir vērtību tipi un glabā reālus datus, atmiņa kuriem tiek izdalīta programmas stekā. Bet klases ir norādes tipi un klases tipa mainīgie glabā tikai norādes uz reāliem datiem. Tāpēc pieeja pie struktūras elementiem notiek ātrāk nekā pie klases elementiem, bet kā funkciju parametrus ir izdevīgāk izmantot klases, tāpēc, kā dotajā gadījumā nenotiek vērtību kopēšana.

Aprakstīt struktūru var ar sekojošu operatoru:

<pieejas_modifikātors> struct <vārds> : <interfeiss>

<struktūras ķermenis>

Piemēram:

```
using System;
```

```
class Usestruct
{
	struct point
	{
		public int x, y;
		public point(int x, int y)
		{
			this.x = x;
			this.y = y;
		}
	}
	public static void Main()
	{
		point [] pt = new point [5];
		for (int i=0; i<5; i++)
		{
			pt[i] = new point(i, i+2);
			Console.WriteLine("point: {0},{1}", pt[i].x, pt[i].y);
		}
	}
}
```

Pirms struktūras vārda un tās elementiem var izmantot sekojošus pieejas modifikatorus:

- **public –** pieeja nav ierobežota;
- **private –** pieeja ir iespējama tikai struktūras robežās, neviens ārējais objekts pie atbilstoša elementa piekļūt nevar;
- **internal –** pieeja ir atļauta tikai objektiem, kas atrodas tajā pat salikumā (assembly).

Piezīme:

Visi vienkāršie (bāzes) tipi, kas tiek lietoti valodā C# ir struktūras. Piemēram, programmā var izmantot operatorus līdzīgus sekojošiem:

int i = 5; Console.WriteLine(i.ToString());

vai:

Console.WriteLine(5.ToString());

Izveidot struktūru var ar operatoru **new**, bet izmantot operatoru **new** nav obligāti. Struktūras inicializācijai var izmantot tikai konstruktorus, bet konstruktoru pēc noklusēšanas piedāvā sistēma. Piemēram:

```
using System;
class Usestruct
{
	struct mystruct
	{
		long l;
		public long LProp
		{
			get
			{
				return l;
			}
			set
			{
				if (value >10)
					l = value;
			}
```

```
            }
            public void Show()
            {
                  Console.WriteLine("l={0}", l);
            }
      }
      public static void Main()
      {
            mystruct ms = new mystruct();
            ms.LProp = 12;
            ms.Show();
      }
}
```

Dotā piemērā tiek izveidota struktūra, kas satur metodi **Show** un īpašību **LProp**.

Uzskaitījumi

Pēc būtības, uzskaitījumi ir veselas konstantes, kurām ir piešķirti ērti nosaukumi, piemēram:

```
enum Languages {English, French, Spanish, German};
```

Šeit identifikatoram **English** tiek piešķirta vērtība 0, identifikatoram **French –** vērtība 1 un t.t. Bet konstanšu vērtības var norādīt arī tieši, piemēram:

```
enum Languages {English = 10, French, Spanish = 20, German};
```

Norādes tipi

Norādes tipa mainīgie glabā nevis reālus datus, bet informāciju par datu glabāšanas vietu. Norādes tipu piemēri ir rindas, masīvi, objekti, klases, interfeisi un delegāti.

Masīvi

Valodā C# visi masīvu tipi ir atvasināti no tipa **System.Array.** Atšķirībā no C valodas, valodā C# aprakstot masīvu nav jānorāda tā izmēru. Piemēram, aprakstīt viendimensiju masīvu var ar operatoru:

```
int [] digits;
```

Pirms masīva elementu izmantošanas, masīvu ir jāizveido ar operatoru **new** un jānorāda tā izmērs, piemēram:

```
digits = new int [15];
```

Dotais operators aizpilda masīvu ar vērtībām pēc noklusēšanas. Bet masīva veidošanas brīdī atļauts arī inicializēt masīva elementus, piemēram:

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

Masīva elementus var arī inicializēt masīva apraksta brīdī:

```
int [] numbers = {1, 3, 5, 7, 9};
```

Dotajā gadījumā vārds **new** nav jānorāda.

Piekļūt pie masīva elementa var norādot tā indeksu kvadrātiekāvās aiz masīva vārda. Pirmā elementa indekss ir 0. Piemēram:

```
int i;
string [] names = new string [10];
for (i=0; i<10; i++)
{
      Console.WriteLine("Input name: ");
      names[i] = Console.ReadLine();
}
```

Divdimensiju masīvu var aprakstīt sekojošā veidā:

```
int [,] mas = new int [10,10];
```

un piekļūt pie tā elementiem var ar operatoru, līdzīgu sekojošam:

```
mas[5,6] = 7;
```

Līdzīgi, trīsdimensiju masīvu var aprakstīt šādi:

```
int [,,] j = new int [2,3,5];
```

Valoda C# var izmantot arī saliktus masīvus (jagged array) vai masīvu no masīviem, kuru izmēri atšķiras. Piemēram, sekojošā programmā tiek veidots saliktais masīvs, kas satur divus masīvus no **int** tipa elementiem.

```
int [][] jmas = new int [2][];
jmas[0] = new int [4] {1, 2, 3, 4};
jmas[1] = new int [3] {1, 4, 9};
for (int i=0; i<2; i++)
{
      Console.WriteLine("Element ({0})", i);
      Console.WriteLine();
      for (int j=0; i<jmas[i].Length; j++)
            Console.WriteLine("   {0}", jmas[i][j]);
}
```

Ja masīvu ir nepieciešams nodod funkcijai kā parametru, tad funkcijas virsraksts var izskatīties sekojošā veidā:

```
static void writeArr(int [] numbers)
```

Masīvu apstrādei var izmantot arī operatoru **foreach**. Piemēram:

```
int [] num = new int [10];
...
foreach (int i in num)
{
      Console.WriteLine("Masīva elements – {0}", i);
}
```

Simbolu virknes (rindas)

Rindu glabāšanai ir paredzēts tips **string** (pilns nosaukums – **System.Srting**). Tieši modificēt rindas ir aizliegts, rindu modificēšanai ir paredzētas klases **string** metodes, tādas kā:

Compare – kas salīdzina divas rindas;
IndexOf – meklē vienu rindu citā rindā;

Concat – saķēdē rindas;
Insert – iestarpina vienu rindu citā;
Remove – dzēš norādītos simbolus no rindas;
ToLower, ToUpper – pārveido burtus mazos un lielos atbilstoši;
Trim – dzēš tukšumzīmes rindas sākumā un beigās;
un t.t.

Bet visas dotās metodes nemodificē eksistējošu objektu, bet dod atpakaļ jaunu objektu ar visām norādītām izmaiņām. Tas nozīmē to, ka ja pierakstīt vienkārši: *str.Remove(4,1);* nevis *str = str.Remove(4,1);* tad nekas nenotiks. **String** klases metodes nemodificē eksistējošās rindas, bet dod atpakaļ jaunas. Tāpēc, ja ir nepieciešams bieži modificēt rindas ir ieteicams izmantot klasi **StringBuilder** klases **String** vietā. Piemēram:

```
1) string n1 = "Fred";
   string n2 = "Smith";
   string ss;
   ss = String.Concat(n1, " ", n2);

2) if (String.Compare(s1, s2) == 0)
      Console.WriteLine("strings are equal");

3) string s1 = "Julian Templeman";
   int fpos = s1.IndexOf("n");
   Console.WriteLine("First occurence of 'n' is at {0}", fpos);
   Console.writeLine("Second occurence of 'n' is at {0}", s1.IndexOf("n",
   fpos+1));

4) string s1 = "abcdefgh";
   Console.WriteLine("{0}",s1.Substring(2,4)); // druka cdef
```

Piekļūt pie atsevišķa rindas simbola var norādot tā numuru (indeksu). Piemēram:

```
string str = "abcdifgh";
char ch = str[4];
```

Bet līdzīgā veidā nav iespējams izmainīt rindas simbolu, operators *str[4] = 'e';* netiks izpildīts.
Izmainīt simbolu var ar operatoru:

```
str = str.Replace('i','e');
```

Dotais operators aizvieto visus simbolus 'i' ar simbolu 'e'. Bet ja ir nepieciešams izmainit tikai vienu simbolu, tad var izmantot operatorus, līdzīgus sekojošiem:

```
str = str.Remove(4,1); // dzēš simbolu 'i'
str = str.Insert(4,"e"); // iestarpina 'e'
```

Dotos operatorus var arī apvienot vienā operatorā sekojošā veidā:

```
str = str.Remove(4,1).Insert(4,"e");
```

Iepriekšējā piemērā, neskatoties uz to, ka tiek lietots tikai viens rindas mainīgais **str**, abu metožu izsaukšana veido jaunas rindas, plus "e" tā ir vēl viena rinda. Tāpēc dažos gadījumos daudz ērtāk pārveidot rindu masīvā, izmainīt masīva elementus un pēc tam masīva pamatā izveidot jaunu rindu. Piemēram:

```
char [] ach = str.ToCharArray();
ach[4] = 'e';
```

```
str = new String(ach);
```

Vēl viens variants, kā var izmainīt rindas simbolu ir apakšrindu saķēdēšana. Piemēram:

```
str = str.Substring(0,4) + "e" +  str.Substring(5);
```

Rindas var saturēt vadības simbolus, tādus, kā valodā C, piemēram: simbolus \0 - Null, \a – skaņas signāls, \t – tabulācija, \n – jauna rinda, \\ - slīpsvītra un citus. Lai apgrieztā slīpsvītra netiktu uzskatīta kā vadības simbola daļa, pirms simbolu virknes var pierakstīt simbolu @. Piemēram:

```
string str = @"c:\temp\myfile";
```

Lai iekļautu tādā rindā pēdiņas, tās ir jāpieraksta divas reizes.

Analizēt atsevišķus rindas simbolus var ar **Char** tipa tādām metodēm kā: **IsDigit, IsLetter, IsUpper, IsLower, IsLetterOrDigit, IsPunctuation** un citām. Visas dotās metodes dod atpakaļ loģisku tipa **bool** vērtību, piemēram:

```
if (Char.IsControl(str[iIndex])) .....    vai
if (Char.IsControl(str, iIndex)) .....
```

Pārcilāt visus simbolus rindā var ar operatoriem **for** vai **foreach**. Piemēram:

```
foreach (char ch in str)
{
      ....
      // mainīga ch vērtību var tikai lasīt
}
```

vai

```
for (int i=0; i<str.Length; i++)
{
      char ch = str[i];
      ....
}
```

Inicializēt rindas apraksta brīdī nav obligāti, piemēram:

```
string str1;
```

String tipa mainīgiem var piešķirt vērtību **null**, tāpēc ka **string** ir norādes tips, piemēram:

```
string str2 = null;
```

Dotajā gadījumā rindai netiek izdalīta atmiņa. Bet var arī veidot tukšas rindas:

```
string str3 = "";
```

Piezīme:

str3.Length *dod atpakaļ 0, bet operators* ***str2.Length*** *noved pie izņēmuma situācijas ģenerācijas programmas izpildes laikā.*

Rindas var arī veidot pārveidojot kādu objektu rindā, piemēram:

```
str = 55.ToString();
// mainīgam str tiks piešķirta vērtība "55".
```

Rindu kopēšanai var izmantot piešķires zīmi, piemēram:

```
string strCopy = str;
```

Bet rindu saķēdēšanai operāciju +

```
string str = str1 + str2;
```

Rindu salīdzināšanai var izmantot operatorus == un !=, kas atšķir lielos burtus no maziem. piemēram:

```
if (str == "New York") ....
```

Leksisku rindu salīdzināšanu var izpildīt izmantojot nestatisku klases **String** metodi **CompareTo** un operācijas > un <. Leksiskā salīdzināšana ir jūtīga burtu reģistram tikai ja rindas ir pilnīgi identiskas izņemot reģistru. Dotajā gadījumā mazie burti tiek uzskatīti par mazākiem nekā lielie. Piemēram, operators **"the" < "The"** dod atpakaļ **true**, bet operators **"Them" < "then"** arī dod atpakaļ **true.**

Metodi **CompareTo** var izsaukt šādi:

```
str1.CompareTo(str2)
```

Šī metode dod atpakaļ **int** tipa vērtību.

Operācijas

Valodas C# operāciju zīmes gandrīz pilnībā sakrīt ar valodas C++ operāciju zīmēm. Operāciju zīmes valodā C# sauc par operatoriem. Operatoru prioritātes ir sekojošas:

1. **prioritāte ir sekojošiem operatoriem:**
   - **()** iekavas
   - **.** piekļūšana pie klases vai struktūras elementa
   - **()** funkcijas izsaukšana
   - **[]** piekļūšana pie masīva elementa
   - **++, --** inkrementa un dekrementa operācijas
   - **new** objekta izveidošanas operācija
   - **tipeof** tipa noteikšanas operācija
   - **sizeof** izmēra noteikšana
   - **checked** pārpildināšanas pārbaudīšana
   - **unchecked** pārpildināšanas pārbaudīšanas atcelšana
2. **prioritāte ir sekojošiem operatoriem:**
   - unāras **+** un **–**
   - **!** loģiskā negācija
   - **~** bitsecīgā negācija
   - **()** tipa pārveidošana
3. **prioritāte ir sekojošiem operatoriem:**
   - ***** reizināšana
   - / dalīšana
   - **%** dalīšanas atlikums
4. **prioritāte ir sekojošiem operatoriem:**
   - **+** saskaitīšana
   - **-** atņemšana
5. **prioritāte ir nobīdes operācijām:**
   - >> un <<
6. **prioritāte ir attiecību operācijām:**
   - **>, <, >=, <=, is** (tipu savietojamība)
7. **prioritāte ir vienādības operācijām:**

== vienāds
!= nav vienāds

8. **prioritāte ir bitsecīgām operācijām:**
   & un
   ^ izslēdzoša vai
   | vai
9. **prioritāte ir loģiskām operācijām (ir atļautas tikai ar tipu bool):**
   && un
   || vai
10. **prioritāte ir pārbaudes operācijai ?:**
11. **prioritāte ir piešķires operācijām:**
    =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Programmas vadības operatori

1. Nosacījuma operatora **if** formāts ir tieši tāds, kā valodā C, piemēram:

```
if (i>10)
{
      b = true;
}
else
{
      b = false;
}
```

2. Izvēles operators **switch** atšķiras no atbilstošā valodas C operatora ar to, ka kā **case** iezīmes var izmantot rindas, piemēram:

```
string s;
s = Console.ReadLine();
switch (s)
{
      case "monday":
      case "Monday":
            Console.WriteLine("Jūs izvēlas Monday");
            break;
      case "tuesday":
      case "Tuesday":
            Console.WriteLine("Jūs izvēlas Tuesday");
            break;
      default:
            Console.WriteLine("Jūs ievadījat nepareizu vārdu");
            break;
}
```

3. Cikla operatoru **for**, **while** un **do-while** formāti sakrīt ar valodas C atbilstošiem operatoriem, piemēram:

a)
```
for (i=0; i!=10; i++)
{
      Console.WriteLine(i);
}
```

b)
```
string ans = "y";
```

```
        while (ans != "n")
        {
              Console.WriteLine("Input Number");
              string s = Console.ReadLine();
              int i = Int32.Parse(s);
              int square = i*i;
              Console.WriteLine("square = {0}", square);
              Console.WriteLine("Continue?(y/n)");
              ans = Console.ReadLine();
        }
```

c)
```
        string ans = "y";
        do
        {
              Console.WriteLine("Input Number");
              string s = Console.ReadLine();
              int i = Int32.Parse(s);
              int square = i*i;
              Console.WriteLine("square = {0}", square);
              Console.WriteLine("Continue?(y/n)");
              ans = Console.ReadLine();
        } while (ans != "n");
```

4. Ciklu var organizēt arī ar operatora **foreach** palīdzību. Operators **foreach** ir paredzēts masīvu un kolekciju elementu pārcilāšanai. Piemēram:

```
int []a = {1, ,3, 0, 2, 4};
foreach (int i in a)
{
      Console.WriteLine(i);
}
```

   Dota programma izvada masīva elementus ekrānā.

5. Pārējas operatoru **goto, break, continue, return** formāti ir tieši tādi, kā valodā C.

6. Izņēmumu situācijas apstrādei ir paredzēti operatori **try** un **throw.** Valoda C# nodrošina daudz standartizņēmumus, piemēram: **System.OutOfMemoryException** (nav iespējams ar **new** izdalīt atmiņu), **System.StackOverflowException** (parādās kad steks ir piepildīts, kad steks satur pārāk lielu metožu izsaukumu skaitu), **System.IndexOutOfRangeException** (parādās kad programmētājs mēģina piekļūt pie neeksistējoša masīva elementa, piemēram ar negatīvu indeksu) un t.t.

   Visi izņēmumu tipi ir tipa **System.Exception** pēcteči un visiem izņēmumu tipiem ir īpašība **Message**, kas satur informāciju par izņēmuma parādīšanas iemeslu.

   Izveidot izņēmuma situāciju programmētājs var ar operatoru **throw <izņēmuma tips>**. Piemēram:

```
throw (new System.ArrayTypeMismatchException());
```

   Kā izņēmuma tipu aiz operatora **throw** var izmantot jebkuru tipu, kas ir klases **System.Exception** pēctecis (tajā skaitā arī tipus, kurus izveidoja pats programmētājs uz klases **System.Exception** bāzes) vai pašu tipu **System.Exception**.

   Izņēmumu apstrādei var izmantot operatoru **try-catch-finally**, kuram ir sekojoša sintakse:

   try
   <kods, kas var izraisīt kļūdu>
   catch
   <operatori, kurus ir jāizpilda, ja parādījās kļūda>
   finally

<operatori, kurus ir jāizpilda neatkarībā no tā, vai bija izraisīta kļūda vai nē>

Operators **try** var saturēt vairākus **catch** blokus, paredzētus kļūdu atlasei. Šajā gadījumā aiz **catch** ir jānorāda izņēmuma tipu, kuru ir jāapstrādā atbilstošam **catch** blokam. Lai bloks **catch** apstrādātu visas iespējamās kļūdas, tad to ir jāpieraksta sekojošā veidā:

```
catch (Exception  ex)   // vai catch (System.Exception ex)
```

un šo bloku ir jāpieraksta pēdēju. Piemēram:

```
using System;
class BooksExcep
{
      static void Main()
      {
            Console.WriteLine("Please enter the price in dollars");
            string price = Console.ReadLine();
            int iprice = Int32.Parse(price);
            int i = 0;
            try
            {
                  int div = iprice/i;
            }
            catch (DivideByzeroException d)
            {
                  Console.writeLine("The {0} exception is caught ", d);
            }
            catch (Exception ex)
            {
                  Console.writeLine("The {0} exception is caught ", d);
            }
            finally
            {
                  Console.writeLine("iprice={0} i={1}", iprice, i);
            }
      }
}
```

Piezīmes:

1) *Norādīt bloku **finally** nav obligāti.*
2) *Ja operators **throw** atrodas blokā **catch**, tad aiz tā atļauts norādīt tikai to izņēmumu, kas ir norādīts aiz **catch**. Piemēram:*

```
catch (OverflowException o)
{
      throw (o);
}
```

7. Operatori **checked** un **unchecked** ir paredzēti pārpildināšanas pārbaudīšanai. Operatoriem ir sekojoša sintakse:

   [un]checked (<izteiksme>) vai
   [un]checked {<bloks>}

   Ja izteiksmes vērtības aprēķināšanas laikā notika pārpildināšana un izteiksme atrodas aiz operatora **checked**, tad parādās izņēmuma situācija. Bet ja izteiksme atrodas aiz operatora **unchecked** un parādās pārpildināšana, tad vecāki izteiksmes vērtības biti tiek ignorēti.
   Ja parādās pārpildināšana un neviens no operatoriem **checked** un **unchecked** nav norādīts, tad viss ir atkarīgs no vides konfigurācijas un kompilatora parametriem.

Klases

Klases pieder pie norādes tipiem un satur tādus datus kā īpašības, metodes un notikumus. Aprakstīt klasi var ar sekojošu izteiksmi:

<pieejas modifikators> class <nosaukums> : <bāzes klases nosaukums>
{ <klases ķermenis> }

Atšķirība no C++ valodas klases apraksta beigās nav obligāti rakstīt semikolu (;). Valodā C#, tapāt kā valodā Java, klases realizācija netiek atdalīta no klases apraksta.

Modifikatori

Valodā C# atļauts izmantot sekojošus pieejas veida modifikatorus: **new, public, protected, internal, private, abstract, sealed.**

Modifikatoru **new** var norādīt tikai klasēm, kas ir aprakstītas citas klases iekšā. Modifikators **new** tiek lietots lai paslēptu klasi, mantotu no bāzes klases. Piemēram:

```
using System;
public class Base1
{
      public class Nested
      {
            public int a =25;
      }
}
public class MyDerived : Base1
{
      new public class Nested
      {
            public int a = 43;
      }
      public static void Main()
      {
            Nested s1 = new Nested();
            Base1.Nested s2 = new Base1.Nested();
            Console.WriteLine(s1.a); // 43
            Console.WriteLine(s2.a); // 25
      }
}
```

Modifikators **public** atļauj jebkuram objektam, tajā skaitā objektam, kas atrodas ārpus no tekošā salikuma, piekļūt pie klases.

Modifikatoru **protected** var norādīt tikai pirms tādām klasēm, kas ir citas klases elementi (par kādas klases elementu var kalpot arī cita klase). Modifikators **protected** nozīmē, ka atbilstošais klases elements ir pieejams tikai tekošā klasē un tās pēctečos.

Modifikators **internal** nozīmē, ka klase ir pieejama tikai tajā salikumā, kurā atrodas klases apraksts. Ja klases aprakstā nav norādīts neviens no modifikatoriem, tad tiek uzskatīts, ka klasei ir pieejas veids **internal.**

Modifikatoru **private** var norādīt tikai pirms tādām klasēm, kas ir citas klases elementi (par kādas klases elementu var kalpot arī cita klase), tas nozīmē, ka atbilstošie elementi ir pieejami tikai klases iekšienē.

Modifikators **abstract** norāda to, ka klasi var izmantot tikai kā bāzes klasi citu klašu veidošanai un dotās klases pamatā nedrīkst veidot objektus. Piemēram:

```
using System;
public abstract class Items
{
      protected int item_code;
      protected float price;
      protected string name;
}
public class Books : Items
{
      private int no_of_pages;
}
public class CD : Items
{
      private int playing_time;
}
public class MyClass
{
      public static void Main()
      {
            Books b1 = new Books();
            CD cd1 = new CD();
            Items i1 = new Items()
      // kļūda!!! abstraktu klasi nevar realizēt!
      }
}
```

Modifikators **sealed** norāda to, ka klasi nedrīkst izmantot kā bāzes klasi citu klašu veidošanai.

Piezīme:

Ir aizlēgts vienlaicīgi izmantot modifikatorus ***sealed un abstract.***

Klases elementi

Klases var saturēt sekojošus elementus: konstantes, laukus, metodes, īpašības, notikumus, indeksatorus, operatorus, konstruktorus un destruktorus.

Konstantes

Konstantes ir paredzētas patstāvīgu vērtību glabāšanai. Šīs vērtības ir aizliegts mainīt programmas izpildes gaitā. Piemēram:

```
public class Books
{
      public const string sort_code = "12-00-23";
      ....
}
```

Kā modifikatora **const** alternatīvu var izmantot modifikatoru **readonly,** kas atšķiras no **const** ar to, ka klases elementam tipa **readonly** var piešķirt sākumvērtību klases konstruktorā. Piemēram:

```
public class Books
{
      public readonly string sort_code;
      public Books(string code)
      {
            sort_code = code;
      }
      ....
}
```

Lauki

Lauki pēc savas būtības ir mainīgie, kas ir saistīti ar klasi vai objektu. Lauku apraksta piemērs:

```
class MyClass
{
      public static int num1=1, num2;
      ....
}
```

Pirms lauka tipa var norādīt vienu no sekojošiem modifikatoriem: **new, public, protected, internal, private, static, readonly.**

Modifikators **static** nozīmē to, ka neatkarīgi no klases eksemplāru skaita tiek veidots tikai viens statiskais lauks visiem klases eksemplāriem. Pārējie modifikatori ir minēti agrāk, tos var norādīt arī pirms vārda **class**.

Metodes

Metodes realizē darbības, kuras izpilda klase vai objekts. Metodes ir jāapraksta ar sekojošu izteiksmi:

[<modifikatori>] < tips> <metodes nosaukums>(<parametru saraksts>)
{
<metodes ķermenis>
}

Aprakstot metodes ir atļauts izmantot sekojošus modifikatorus: **new, public, protected, internal, private, static, virtual, override, extern, abstract.**

Modifikators **virtual** tiek lietots polimorfizma nodrošināšanai. (Polimorfizms tā ir iespēja katrā klasē realizēt vienu un to pašu metodi savādāk).
Piemēram:

```
// 1. piemērs bez vārda virtual
using System;
public class Items
{
      public void display()
      {
            Console.WriteLine("Items");
      }
      ....
}
public class Books : Items
```

```
{
      public new void display()
      {
            Console.WriteLine("Books");
      }
      ....
}
public class MainClass
{
      public static void Main()
      {
            Items i1 = new Books();
            i1.display();     // izvada "Items"
      }
}
// 2. piemērs ar vārdu virtual
using System;
public class Items
{
      public virtual void display()
      {
            Console.WriteLine("Items");
      }
      ....
}
public class Books : Items
{
      public override void display()
      {
            Console.WriteLine("Books");
      }
      ....
}
public class MainClass
{
      public static void Main()
      {
            Items [] i1 = new Items[20];
            i1[0]  new Books();
            i1[0].display();  // izvada "Books"
      }
}
```

Modifikators **override**, kā ir redzams no iepriekšējā piemēra, norāda uz to, kā klasē ir savādāk realizēta bāzes klases metode.

Modifikators **new** – norāda uz to, ka pēcteča klases metode ir pilnīgi jauna un paslēpj nevis pārdefinē bāzes klases metodi ar tieši tādu vārdu.

Modifikators **extern** norāda to, ka metodes realizācijas kods atrodas ārpus klases robežām un nav pierakstīts valodā C#. Tādas metodes sauc par ārējām metodēm. Modifikatoru **extern** bieži izmanto, lai aprakstītu metodes, kas atrodas ārējās bibliotēkas DLL. Piemēram:

```
using System;
using System.Runtime.InteropServices;
class externModifierDemo
{
      [DllImport("User32.dll")]
```

```
        public static extern int MessageBox(int i, string j, string k, int l);
        public static int Main()
        {
              string str;
              Console.Write("Input Message:");
              str = Console.ReadLine();
              return MessageBox(0, str,
                    "Function MessageBox is called from User32.dll",0);
        }
}
```

Pievērsiet uzmanību tam, ka metode **MessageBox** ir tikai aprakstīta, tās ķermenis nav norādīts.

Piezīmes:

1) *Ir aizliegts vienlaicīgi izmantot modifikatorus **extern** un **abstract.***
2) *Ja metodes aprakstā tiek lietots atribūts **DllImport**, tad obligāti ir jāizmanto modifikatoru **static.***

Modifikators **abstract** tiek lietots metodes aprakstā, ja metode ir realizēta klases pēctečos, bet tekoša klase nesatur metodes realizāciju.

Piezīmes:

1) *Abstraktās metodes var atrasties tikai klasēs, aprakstītas ar modifikatoru **abstract.***
2) *Aizliegts vienlaicīgi izmantot modifikatorus **abstract** un **virtual**. Abstraktā metode jau ir virtuāla.*

Valodā C# var aprakstīt metodes ar mainīgu parametru skaitu. Dotajam mērķim ir paredzēts modifikators **params** kuru ir jāpieraksta pirms metodes parametra. Metodes parametrs tipa **param** pēc savas būtības ir viendimensiju masīvs. Modifikatoru **params** ir aizliegts izmantot kopā ar modifikatoriem **out** un **ref**. Metodei var būt tikai viens parametrs tipa **params.** Piemēram:

```
using System;
class useparam
{
      static void argno(params string [] args)
      {
            Console.WriteLine("Parametru skaits ir {0}", args.Length);
            for (int i=0; i<args.Length; i++)
            {
                  Console.WriteLine("Args({0})={1}", i, args[i]);
            }
      }
      public static void Main()
      {
            argno();
            argno("one", "two");
            argno("one", "two", "three", "four");
            argno(new string[]{"first", "second","third"});
      }
}
```

Konstruktori

Konstruktori ir paredzēti objektu inicializācijai. Konstruktori tiek izsaukti automātiski objekta veidošanas brīdī. Konstruktora vārdam ir jāsakrīt ar klases vārdu. Konstruktors nevar dod atpakaļ kādu vērtību. Konstruktorus var pārlādēt (var izveidot vairākus konstruktorus ar dažādiem parametriem). Piemēram:

```
public class Books
{
      private float price;
      public Books()
      {
            price = 0;
      }
      public Books(float price)
      {
            this.price = price;
      }
      public float GetPrice()
      {
            return price;
      }
}
....
Books b1 = new Books();
Books b2 = new Books(20);
```

Tādus konstruktorus, kā izskatītā piemērā, sauc par eksemplāru konstruktoriem. Valodā C# eksistē arī statiskie konstruktori, kas ir aprakstīti ar modifikatoru **static**. Statiskie konstruktori tiek lietoti, lai inicializētu statiskos klases laukus. Katrai klasei statiskais konstruktors tiek izsaukts tikai vienu reizi, jau pirms atbilstošās klases eksemplāra izveides.

Piezīme:

*Ja konstruktoru aprakstītu ar modifikatoru **private**, tad uz šīs klases bāzes nebūs iespējams veidot atvasinātas klases, kā arī nebūs iespējams izveidot dotās klases objektu. Tāpēc **private** tipa konstruktori tiek lietoti tikai klasēs, kas satur tikai statiskos elementus.*

Destruktori

Destruktori tiek lietoti atmiņas atbrīvošanai. Destruktorus izsauc drazu savācējs pirms objekta iznīcināšanas. Destruktora vārdam ir jāsastāv no simbola ~ un klases vārda. Destruktoram nedrīkst nodod parametrus. Destruktors nevar dot atpakaļ it ne kādu vērtību. Klasē var būt aprakstīts tikai viens destruktors. Piemēram:

```
public class Books
{
      ~Books (){....}
      ....
}
```

Tieši izsaukt destruktorus ir aizliegts. Destruktori tiek mantoti. Pēc darba pabeigšanas destruktors izsauc bāzes klases destruktoru.

Drazu savākšanas pamatproblēma ir saistīta ar to, ka nav iespējams precīzi pareģot, kad drazu savācējs atbrīvos atmiņu un izsauks destruktorus. Parasti tas notiek, kad atmiņas pietrūkst. **Rezultātā var sastapt situācijas, kad destruktors vispār netiks izsaukts.** Programmas aizvēršanas brīdī CLR var izlaist destruktoru izpildīšanu, lai paātrinātu programmas pabeigšanu. Tāpēc nav ieteicams izpildīt destruktorā kādas svarīgas darbības. Tādas darbības, kā failu aizvēršana, datu bāzes ierakstu atjaunošana, ir ieteicams izpildīt atsevišķā metodē, kuru izsaukt tieši. Parasti tādai metodei tiek piešķirts vārds **Dispose**. Šo metodi ir pieņemts izsaukt arī destruktorā un tāpēc metodi **Dispose** ir jāpieraksta tādā veidā, lai tā neģenerētu izņēmumus arī tad, kad tiek izsaukta vienam objektam vairākas reizes.

Piezīmes:

1) *Lai programmas izpildīšanas beigu posmā, visu objektu destruktori tiktu izpildīti ir jāizsauc metode **System.GC.RequestFinalizeOnShutdown**, bet šajā gadījumā programma tiks aizvērta lēnāk.*
2) *Ja pēc metodes **Dispose** darba izsaukt destruktoru vairs nav nepieciešams, tad atcelt destruktora izpildi var ar metodi **System.GC.SupressFinalize** (šo metodi parasti izsauc metodes **Dispose** beigās).*

Īpašības

Īpašības ir analoģiskas laukiem, bet īpašību glabāšanai netiek izdalīta atmiņa. Piekļūšanai pie īpašībām tiek lietotas speciālas procedūras (accessors), kas nosaka izteiksmes, kuras ir jāizpilda īpašības vērtības lasīšanas vai rakstīšanas laikā. Īpašības var aprakstīt ar izteiksmi:

<modifikatori> <tips> [<interfeiss>] identifikators <pieejas funkcijas>

Īpašībām var piešķirt sekojošas pieejas modifikatorus: **new, protected, private, public, internal, static, virtual, abstract** un **override**.

Īpašības vērtības lasīšanai ir paredzēta pieejas funkcija **get**, bet rakstīšanai **set**. Piemēram:

```
public class Book
{
      private string title;
      public string Title
      {
            get
            {
                  return title;
            }
            set
            {
                  title = value;
            }
      }
}
```

Piezīmes:

1) *Funkciju **get** ir jābeidz ar operatoru **return** vai **throw**.*

2) *Ja īpašības aprakstā tiek lietots vārds **abstract**, tad funkciju **get** un **set** ķermeņiem ir jāsastāv no semikola (;). Piemēram:* ***get; set;***
3) *Abstraktās īpašības var aprakstīt tikai abstraktā klasē.*
4) *Ja ir nodefinēta tikai funkcija **get,** tad īpašība ir paredzēta tikai lasīšanai, bet ja tikai funkcija **set**, tad tikai rakstīšanai.*

Indeksatori

Indeksatori dod iespēju apstrādāt objektus, kā masīvus. Aprakstīt indeksatoru var ar sekojošu izteiksmi:

<modifikatori> <tips> this [parametru saraksts] {<pieejas funkcijas>}

Aprakstot indeksatoru, atļauts izmantot sekojošus modifikatorus: **new, public, protected, internal, private, virtual** un **override.** Piemēram:

```
using System;
class BookIndexer
{
      private int [] Bookcollection = new int [50];
      public int this [int index]
      {
            get
            {
                  if (index<0 || index >=50)
                        return 0;
                  else
                        return Bookcollection[index];
            }
            set
            {
                  if (!(index<0 || index>=50))
                        Bookcollection[index] = value;
            }
      }
}
public class MainClass
{
      public static void Main()
      {
            BookIndexer b = new BookIndexer();
            b[3] = 256;
            for (int i=0; i<=3; i++)
            {
                  Console.WriteLine("Book Collection #{0}={1}", i, b[i]);
            }
      }
}
```

Vienā klasē var nodefinēt vairākus indeksatorus, kas atšķiras ar parametriem. Piemēram:

```
class Indexer
{
      ....
```

```
        public string this [int index]
        {
              ....
        }
        public string this [string str]
        {
              ....
        }
}
```

Delegāti

Izpildes vide CLR piedāvā speciālus objektus, kurus sauc par delegātiem. Delegāti var dinamiski izsaukt cita objekta metodes. Pēc savas būtības delegāti ir drošie rādītāji uz funkcijām. Delegātus ir ērti izmantot, kad ir vajadzīgs starpnieks starp izsaucošu procedūru un izsaucamu procedūru. Delegāti arī tiek lietoti, lai saistītu notikumus ar notikumu apstrādes procedūrām. Visi delegāti ir atvasināti no tipa **System.Delegate**. Delegātu var aprakstīt ar sekojošu izteiksmi:

<modifikatori> delegate <atgriežamās vērtības tips> <nosaukums> (<parametri>)

Aprakstot delegātus var izmantot sekojošus modifikatorus: **new, public, protected, internal, private**. Piemēram:

```
public delegate void Newdel();
```

Pēc delegāta apraksta delegātu ir jāinicializē. Piemēram:

```
class usemeth
{
      static void NewMeth()
      {
            Console.WriteLine("Tas ir jauna metode");
      }
}
Newdel n = new Newdel(usemeth.NewMeth);
```

Lai izsauktu metodi ar izveidotā iepriekšējā piemērā delegāta palīdzību var izmantot operatoru **n();** Delegātus var saistīt ne tikai ar statiskām metodēm, bet arī ar klases eksemplāru metodēm. Piemēram:

```
class MyClass
{
      public void mymeth()
      { .... }
      ....
}
....
public delegate void mydel();
....
MyClass obj = new MyClass();
mydel d = new mydel(obj.mymeth);
d();
```

Delegātus var apvienot, izmantojot operāciju +. Ja izsaukt saliktu delegātu, iegūtu "+" operācijas rezultātā, tad tiek izsauktas visas metodes, kas ir saistītas ar delegātiem, kas pieder saliktam delegātam. Metodes tiek izsauktas tādā kārtā, kādā tās tika apvienotas saliktā delegātā.
Izslēgt delegātu no salikta delegāta var ar operāciju "-".

Piezīme:

*Apvienot var tikai **void** tipa delegātus, ar vienādiem parametriem. Nedrīkst apvienot delegātus, kuriem ir tipa **out** parametri.*

Piemēram:

```
public delegate void Catdel();
class MyClass
{
      ....
      public static void f1()
      { .... }
      public static void f2()
      { .... }
      public static void Main()
      {
            Catdel l = new Catdel(f1);
            Catdel m = new Catdel(f2);
            Catdel r;
            r = l+m;
            ....
            r();
      }
}
```

Notikumi

Notikumi tiek lietoti, lai paziņotu par kādām darbībām. Objektu, kas izraisa notikumu, sauc par notikuma avotu, bet objektu, kas reaģē uz notikumu, sauc par notikuma saņēmēju. Lai saistītu notikuma avotu ar notikuma saņēmēju valodā C# tiek lietoti delegāti, kurus sauc par notikumu apstrādātājiem.
Lai programmā izmantotu notikumus ir nepieciešams:

1. izveidot delegātu;
2. izveidot klasi, kas ir notikumu avots un dotajā klasē:
    a. ar vārda **event** palīdzību ir jāapraksta notikumu, kas pēc savas būtības ir delegāts
    b. jānodefinē metodes, kas izraisa notikumu
3. izveidot notikumu saņēmēja klasē, kurā aprakstīt metodes, paredzētas notikumu apstrādei.

Piemēram:

```
using System;
public delegate void EventDemoDel();
public class Share
{
      public event EventDemoDel MyEvent;
      public void Method1()
      {
            if (MyEvent != null)
                  MyEvent();
      }
```

```
}
public class Class1
{
      static private void MyEventHandler()
      {
            Console.WriteLine("Event occured");
      }
      static public void Main()
      {
            Share a1 = new Share();
            a1.MyEvent +=new EventDemoDel(MyEventHandler);
            a1.Method1();
      }
}
```

Dotajā piemērā, sākumā tiek aprakstīts delegātu tips **EventDemoDel**. Pēc tam, klasē **Share** tiek aprakstīts notikums **MyEvent** tipa **EventDemoDel** un metode **Method1,** kas ģenerē notikumu **MyEvent** ar operatoru **MyEvent();**. Lauks **MyEvent** var būt vienāds ar **null**, ja pie notikuma nav piesaistīta it ne viena notikumu apstrādes funkcija. Tāpēc pirms notikuma izsaukšanas ar operatoru **if (MyEvent != null)** tiek pārbaudīts, vai eksistē šī notikuma apstrādes funkcija. Klasē **Class1** ir aprakstīta notikumu apstrādes funkcija **MyEventHandler(),** kas izvada ekrānā paziņojumu. Šī funkcija tiek piesaistīta pie atbilstoša notikuma ar operatoru:

```
a1.MyEvent +=new EventDemoDel(MyEventHandler);
```

Operatori

Valodā C# var pārdefinēt standartas operācijas, tādas kā +, -, ++, !, /, >> un t.t. Nedrīkst pārdefinēt tikai operāciju = un operāciju [] (pieeja pie masīva elementa, bet šos operāciju "pārdefinēšanai" var izmantot indeksatorus). Pārdefinēt operāciju var ar vārda **operator** palīdzību. Piemēram:

```
class Point
{
      public int x, y;
      public Point(int xx, int yy)
      {
            x = xx; y = yy;
      }
      public static Point operator +(Point a, point b)
      {
            return (new Point(a.x+b.x, a.y+b.y));
      }
}
....
Point a = new Point(3,5);
Point b = new Point(6,8);
point c = a+b;
```

Interfeisi

Interfeiss ir norādes tips, kas satur tikai abstraktus elementus. Par interfeisa elementiem var kalpot notikumi, metodes, īpašības un indeksatori. Interfeisā nedrīkst nodefinēt konstantes, laukus, konstruktorus, destruktorus un statiskus elementus.

Interfeiss apraksta standartu, kuram ir jāatbilst atbilstošai klasei. Interfeisu var aprakstīt ar sekojošu izteiksmi:

<modifikatori> interface <nosaukums> : <bāzes interfeisi (atdalīti ar komatiem)>
{ <ķermenis> }

Piemēram:

```
public delegate void DemoEvent(Idemo sender);
public interface Idemo
{
      void SomeFunc(int i);
      int SomeProperty { set; }
      event DemoEvent SomeEvent;
      int this [int index] {get; set; }
}
```

Aprakstīt klasi, kas realizē interfeisu var tā:

```
class DemoClass : Idemo
{
      public void SomeFunc(int i) { .... }
      public int SomeProperty { set { .... } }
      public int this [int index] {get { .... } set { .... } }
      public event DemoEvent SomeEvent;
      ....
}
```

Struktūras arī var realizēt interfeisus.

Klasei, kas realizē interfeisu, obligāti ir jārealizē visas atbilstošā interfeisa metodes.

Pirms vārda **interface** var izmantot sekojošus modifikatorus: **new** (tikai iekļautiem kādā citā interfeisā interfeisiem), **public, protected, internal, private.** Bet pirms interfeisa elementiem aizliegts norādīt sekojošus modifikatorus: **abstract, protected, public, internal, private, virtual, override, static.** Visiem interfeisa elementiem ir jābūt pieejamiem visur.

Aprakstot intefeisu var norādīt vienu vai vairākus bāzes interfeisus. Piemēram:

```
interface First
{
      void Add();
}
interface second
{
      void Delete();
}
interface Third : First, Second { .... }
```

Realizēt interfeisa metodes kādā klasē var tieši vai netieši. Tiešas realizācijas gadījumā pirms metodes vārda ir jānorāda interfeisa vārdu atdalot tos ar punktu.

Piezīme:
Ja metode ir realizēta tieši, tad piekļūt pie dotās metodes var tikai ar interfeisa eksemplāru.

Piemēri:

```
//netieša interfeisa realizācija
```

```
using System;
interface Inter1
{
      void Method1();
}
interface Inter2
{
      void Method2();
}
class Class1 : Inter1, Inter2
{
      public void Method1()
      { .... }
      public void Method2()
      { .... }
      public static void Main()
      {
            Class1 c1 = new Class();
            c1.Method1();
            c1.Method2();
      }
}

//tieša interfeisa realizācija
using System;
interface Inter1
{
      void Method1();
}
interface Inter2
{
      void Method2();
}
class Class1 : Inter1, Inter2
{
      void Inter1.Method1()
      { .... }
      void Inter2.Method2()
      { .... }
      public static void Main()
      {
            Class1 c1 = new Class();
            c1.Method1();     // Kļūda!!!
            Inter1 i1 = (Inter1)c1;
            i1.Method1();     // OK
            Inter2 i2 = (Inter2)c1;
            i2.Method2();     // OK
      }
}
```

Ja klase realizē divus interfeisus un abos interfeisos ir metodes ar vienu un to pašu nosaukumu un ar vienādiem parametru sarakstiem, tad realizēt šīs metodes atbilstošā klasē var tikai izmantojot tiešu realizāciju. Piemēram:

```
interface Inter1
{
      void Method1();
}
```

```
interface Inter2
{
      int Method1();
}
class Class1 : Inter1, Inter2
{
      void Inter1.Method1()
      { .... }
      int Inter2.Method1()
      { .... }
      ....
}
```

Piezīme:

Ja tiek izmantota tieša realizācija, tad aiz klases vārda obligāti ir jānorāda atbilstošā interfeisa nosaukums, piemēram:

```
interface MyInterface
{
      void Method1();
}
class MyClass : MyInterface
{
      void MyInteface.Method1() { .... }
      int OtherInterface.OtherMethod() { .... } // Kļūda !!!
}
```

Ir iespējams piesaistīt vienu klases metodi pie vairākām vienādām metodēm, aprakstītām dažādos interfeisos. Piemēram:

```
interface Inter1
{
      int Method1();
}
interface Inter2
{
      int Method1();
}
class MyClass : Inter1, Inter2
{
      public int Method1()
      { .... }
}
```

Klases pēctecis manto interfeisa realizāciju no bāzes klases. Bet atbilstošas bāzes klases metodes var pārdefinēt, vai atvasinātā klasē atkārtoti realizēt interfeisu. Piemēram:

```
// 1. piemērs : mantošana
interface Inter1
{
      void  Method1();
}
class MyClass : Inter1
{
      public virtual void Method1()
      { .... }
}
```

```
class MySecondClass : MyClass
{
      public override void Method1()
      { .... }
}
....
MyClass mc = new MyClass();
MysecondClass ms = new MySecondClass();
Inter1 ic = mc;
Inter2 it = ms;
mc.Method1(); // izsauc MyClass.Method1();
ms.Method1(); // izsauc MysecondClass.Method1()
ic.Method1(); // izsauc MyClass.Method1();
it.Method1(); // izsauc MySecondClass.Method1();
```

Bet, ja dotajā piemērā izdzēst vārdu **virtual** un vārdu **override** aizvietot ar vārdu **new**, tad pēdējais operators **it.Method1();** izsauks metodi **MyClass.Method1**.

Piezīme:

*Tiešās interfeisa elementu realizācijas nedrikst aprakstīt ar vārdu **virtual**. Bet no tiem var izsaukt virtuālās metodes, pārdefinētas pēctečos.*

```
// 2. piemērs: Atkārtota interfeisa realizācija
interface Inter1
{
      void Method1();
}
class Class1 : Inter1
{
      void Inter1.Method1() { .... }
}
class Class2 : Class1, Inter1
{
      public void Method1() { .... }
}
```

Piezīme:

Abstraktas klases arī var realizēt interfeisus.

Ar operācijas **is** palīdzību var pārbaudīt, vai klase atbilst kādam interfeisam vai nē. Piemēram:

```
MyClass mc = new MyClass();
if  (mc is Inter1) { .... }
```

Dotajam mērķim var izmantot arī operāciju **as**, kas dod atpakaļ **null,** ja klase neatbilst norādītam interfeisam. Piemēram:

```
Inter1 temp = mc as Inter1;
if (temp != null)
{
      temp.SomeFunc();
}
```

Kolekcijas

Par kolekcijām sauc klases, kas satur un apstrādā līdzīgu objektu grupas. Vārdu telpa **System.Collections** satur klašu un interfeisu realizācijas, kas atvieglo kolekciju izmantošanu programmā. Piemēram, programmā var izmantot sekojošas klases:

- **System.Array –** fiksēta izmēra masīvs
- **ArrayList –** masīvs, kura izmēru var mainīt dināmiski
- **Stack –** elementu saraksts, kas ir izveidots pēc principa LIFO (last in first out)
- **Hashtable –** elementu saraksts, kurā piekļūt pie elementa var ar atslēgas palīdzību
- **SortedList –** piekļūt pie saraksta elementiem var ar indeksa palīdzību
- **Queue –** elementu saraksts, kas ir izveidots pēc principa FIFO (first in first out)
- **BitArray –** karodziņu **true/false** vai **on/off** saraksts
- **StringCollection –** rindu saraksts, piekļūt pie saraksta elementa var izmantojot indeksu
- **NameValueCollection –** rindu saraksts, kurā piekļūt pie elementa var pēc atslēgas vai pēc indeksa

Gandrīz visas kolekcijas, aprakstītas vārdu telpā **Collections,** realizē interfeisu **IEnumerable**, tas nozīmē, ka kolekcijai ir metode **GetEnumerator(),** kas dod atpakaļ objektu-iterātoru, kas realizē interfeisu **IEnumerator** un piedāvā metodes, paredzētas kolekciju elementu pārcilāšanai. Piemēram, tādas metodes, kā **MoveNext (**pāriet pie nākošā elementa), **Reset** (pāriet pie pirmā elementa), **Current** (dot atpakaļ norādi uz tekošo kolekcijas elementu).

Interfeiss **ICollection,** kas ir interfeisa **IEnumerable** pēctecis, nodrošina tādas metodes, kā **CopyTo** (nokopēt kolekcijas elementus viendimēnsiju masīvā) un piedāvā arī īpašību **Count,** ar kuras palīdzību var noteikt elementu skaitu kolekcijā.

Interfeisu **IList**, kas ir interfeisa **ICollection** pēctecis, realizē klases, kas nodrošina pieeju pie kolekcijas elementiem ar indeksa palīdzību. Piemēram, šo interfeisu realizē tādas klases, kā **System.Array** un **System.Collections.ArrayList**. Dotais interfeiss nodrošina sekojošas metodes: **Add, Insert –** elementu pievienošanai, **Remove, RemoveAt –** elementu dzēšanai, **IndexOf, Contains** – elementu meklēšanai, **Item –** piekļūšanai pie elementa pēc indeksa, **IsFixedSize** un **IsReadOnly** - paziņo vai dotās kolekcijas izmērs ir pastāvīgs vai nē un vai dotās kolekcijas elementus atļauts mainīt vai nē.

Piemēram, objektu-iterātoru, kas realizē interfeisu **IEnumerator** var izmantot šādi:

```
int [] ar = {10, 11, 12, 13 };
IEnumerator enm = ar.GetEnumerator();
while (enm.MoveNext() == true)
      Console.WriteLine(enm.Current);
```

Bet izmantot klasi **ArrayList** var sekojošā veidā:

```
ArrayList al = new ArrayList();
al.Add("zero");
al.Add("two");
al.Insert(1, "one");
Console.WriteLine("Capacity={0}, Count={1}, Item1={2}",
      al.Capacity, al.Count, al.Item(1));       // izvada: 16 3 one
```

Piezīme:

*Īpašība **Capacity** dod iespēju noteikt vai izmainīt kolekcijas izmēru, bet **Count** dod atpakaļ elementu skaitu kolekcijā. Ar metodi **TrimToSize** var samazināt kolekcijas izmēru līdz elementu skaitam.*

Programmētājs var veidot savas kolekcijas uz sekojošu klašu bāzes: **CollectionBase**, **ReadOnlyCollectionBase** un **DictionaryBase**, realizējot tajās vajadzīgos interfeisus.

Rezervēti vārdi this **un** base

Vārdi **this** un **base** var būt izmantoti tikai konstruktoros vai nestatiskās metodēs. Vārds **this** dod iespēju piekļūt pie tekošā objekta metodēm, bet vārds **base** – pie bāzes klases elementiem, paslēptiem ar tekošās klases elementiem ar līdzīgiem vārdiem.

Vārdu telpas

Vārdu telpa - tas ir objekts-konteiners, kas tiek lietots klašu, struktūru, interfeisu un delegātu apvienošanai. Vārdu telpas dod iespēju izvairīties no vārdu konfliktiem. Pēc noklusēšanas vārdu telpa tiek veidota automātiski bez speciāla apraksta. Tādu vārdu telpu sauc par globālu. Globālā vārdu telpa ir katrā programmā. Bez tam programmētājs var ar vārdu **namespace** aprakstīt savas vārdu telpas, kas var būt iekļautas viena otrā. Piemēram:

```
namespace BooksOnline
{
      ....
      namespace Scientific
      {
            ....
            class Beginners
      }
}
```

Šajā gadījumā pilns klases **Beginners** nosaukums ir **BooksOnline.Scientific.Beginners.** Lai saīsinātu tādus garus nosaukumus, var izmantot direktīvu **using**. Piemēram:

```
using BooksOnline.Scientific;
```

Ja dotu rindu pierakstīt programmas sākumā, tad piekļūt pie minētās klases būs iespējams izmantojot tikai klases vārdu **Beginners.**

Lai saīsinātu garu nosaukumu var izmantot arī pseidonīmus, kurus var lietot vārdu telpas nosaukuma vietā. Piemēram, aprakstīt pseidonīmu var sekojošā veidā:

```
using BS = BooksOnline.Scientific;
```

Šajā gadījumā piekļūt pie klases **Beginners** būs iespējams ar izteiksmi **BS.Beginners.**

Piezīme:

Divas vārdu telpas ar vienādu pilnu nosaukumu, nodefinētas dažādos failos, tiek uzskatītas kā viena vārdu telpā.

Preprocesora direktīvas

Par preprocesora direktīvām sauc speciālas komandas, kas sākas ar simbolu #. Valodā C# direktīvas tiek lietotas nosacījuma kompilācijas nodrosināšanai. Var izmantot sekojošas direktīvas:

1) **#define**
   Formāts: *#define <simbols>*
   Piemēram:
   ```
   #define SYMB1
   ```
   Tiek lietota simbolu nodefinēšanai. Simbola darbības apgabals tiek ierobežots ar failu, kurā simbols ir nodefinēts.

2) **#undef**
   Formāts: *#undef <simbols>*
   Atceļ simbola nodefinēšanu

3) **#if #else #elif #endif**
   Paredzētas koda daļas izlaišanai kompilācijas laikā. Piemēram:

```
#define SYMBOL1
#undef SYMBOL2
using System;
public class SomeClass
{
      public static void Main()
      {
            #if (SYMBOL1&& SYMBOL2)
                  Console.WriteLine("Abi simboli ir nodefinēti");
            #elif (SYMBOL1 || SYMBOL2)
                  Console.WriteLine(
                        "Ir nodefinēts tikai viens simbols");
            #endif
      }
}
```

4) **#warning**
   Formāts: *#warning <teksts>*
   Izvada brīdinājumu programmas kompilācijas laikā. Parasti tiek lietota nosacījuma kompilācijas bloka iekšā. Piemēram:
   ```
   #warning Example of warning
   ```

5) **#error**
   Formāts: *#error <teksts>*
   Izvada paziņojumu par kļūdu programmas kompilācijas laikā

6) **#region**
   Formāts: *#region <nosaukums>*

kods
#endregion

Dotā direktīva dod iespēju iezīmēt bloku, kuru .NET vides redaktors varēs paslēpt un attēlot pēc peles klikšķiem uz pogām "+" un "-".

Atribūti

Atribūti dod iespēju sasaistīt ar programmas koda elementiem nestandartu informāciju, piekļūt pie kuras būs iespējams programmas izpildes laikā. Atribūtiem nav analogu citās programmēšanas valodās. Atribūtus var norādīt tipu, klašu un interfeisu elementu aprakstā. Visi atribūti ir klases **System.Attribute** pēcteči. Programmētājs var pats nodefinēt jaunus atribūtus, vai izmantot standartus. Atribūtiem var būt parametri. Parametrus var sadalīt divās grupās – pozicionālos un nepozicionālos atribūtos, pieejai pie kuriem tiek lietoti parametru nosaukumi. Piemēram:

```
[AttributeUsage(AttributeTargets.Method)]
class NewAttribute : System.Attribute
{
      public NewAttribAttribute(string date, int no_of_pages) { .... }
      public string category
      {
            get { return category; }
            set { category = value; }
      }
}
class NewClass
{
      [NewAttribAttribute("1/1/2001",20,category="Crime")]
      public static void NMethod()
      {
            Console.WriteLine("This in the NMethod() method");
      }
}
```

Dotā piemērā atribūtam **NewAttibAttribute** ir divi pozicionālie parametri un viens nepozicionālais parametrs **category**. Standarts atribūts **AttributeUsage** norāda, kur var būt izmantots izstrādātais atribūts. Izskatāmā piemērā attributu **NewAttribAttribute** var saistīt tikai ar metodēm.
Sakarā ar to, ka ir pieņemts, ka atribūtu klašu nosaukumi beidzas ar vārdu **Attribute**, piesaistot atribūtu pie koda elementa, vārdu **Attribute** var izlaist. Tas nozīmē, ka izskatāmā piemērā, rindu:

```
[NewAttribAttribute("1/1/2001", 20, category="Crime")]
public static void NMethod()
```

var aizvietot ar rindu

```
[NewAttrib("1/1/2001", 20, category="Crime")] public static void NMethod()
```

Programmas izpildes laikā ar metodi **GetCustomAttributes,** kas pieder vārdu telpai **System.Reflection,** var saņemt informāciju par atribūtiem, nodefinētiem sākotnējā kodā. Metode **GetCustomAttributes** dod atpakaļ masīvu no objektiem, kas ir atribūtu analogi. Piemēram:

```
....
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct)]
public class License : System.Attribute
```

```
{
      public License(int number)
      {
            this.number = number;
            version = 1.0;
      }
      int number;
      public double version;
      public int Get() { return number; }
}
```

Piezīme:

Vārds ***Attribute*** *tiek pievienots pie atribūta nosaukuma automātiski. Tas nozīmē, ka pilns klases* ***License*** *nosaukums ir* ***LicenseAttribute****, bet izsaukt šo atribūtu var izmantojot vārdu* ***License*** *vai* ***LicenseAttribute****.*

```
....
[License(12345, version = 2.5)]
class Some_Class
{
      public void WriteLicense()
      {
            object [] attrs = typeof(Some_Class).GetCustomAttributes();
            for (int i=0; i<attrs.Length; i++)
            {
                  if (attrs[i] is License)
                  {
                        License l = (License)attrs[i];
                        Console.WriteLine("Version={0}, License no {1}",
                              l.version, l.Get());
                  }
            }
      }
}
```

Kā standartu atribūtu piemērus var nosaukt atribūtus:

- **AttributeUsage –** kurš tiek lietots lietotāja atribūtu īpašību norādīšnai
- **conditional –** kurš paredzēts nosacījuma kompilācijas nodrošināšanai un ir direktīvas **#if** analogs
- **dllimport –** kurš tiek lietots, lai norādītu, kur atrodas ārējās metodes realizācijas kods
- **Obsolete –** ar kura palīdzību var atzīmēt vecas programmas daļas, kuras vairs nedrīkst vai nav ieteicams izmantot. Atzīmētas ar **Obsolete** atribūtu metodes izsaukšana noved pie kompilācijas laika kļūdas vai brīdinājuma.

Eksistē arī daudz atribūtu, paredzētu sadarbībai ar COM komponentiem.

Darbs ar failiem

FileStream klase

Klases, paredzētās darbam ar failiem, atrodas vārdu telpā **System.IO**. Viena no galvenajām klasēm, aprakstītām vārdu telpā **System.IO,** ir klase **FileStream**. Ar šīs klases palīdzību var atvērt un aizvērt failu, rakstīt un lasīt informāciju no faila. Pirms kādas operācijas izpildes ar failu ir nepieciešams izveidot klases **FileSream** objektu. Klases **FileStream** konstruktoram ir seši parametri:

1. parametrs - faila nosaukums;
2. parametrs – nosaka, vai ir jāveido jauns fails, ja fails neeksistē. Kā otru parametru klases **FileStream** konstruktoram var nodod vienu no sekojošām konstantēm:
   - **FileMode.CreateNew –** ja fails eksistē, tad parādās kļūda;
   - **FileMode.Create –** ja fails eksistē, tad tā saturs tiks iznīcināts;
   - **FileMode.Open –** ja fails neeksistē, tad parādās kļūda;
   - **FileMode.OpenOrCreate** – ja fails neeksistē, tad tas tiks izveidots;
   - **FileMode**.**Truncate -** kļūda, ja fails neeksistē, faila saturs tiks iznīcināts;
   - **FileMode**.**Append** – kļūda, ja fails tiek atvērts lasīšanai, ja nav, tad tas tiks izveidots, faila radītāis tiek novietots faila beigās;
3. parametrs – nosaka pieejas režīmu. Kā šo parametru konstruktoram var nodod vienu no sekojošām konstantēm: **FileAccess.Read, FileAccess.Write, FileAccess.ReadWrite;**
4. parametrs – nosaka vai citi procesi var vienlaicīgi ar tekošo procesu piekļūt pie atvērtā faila . Kā 4. parametru var izmantot vienu no sekojošām konstantēm: **FileShare.None, FileShare.Read, FileShare.Write, FileShare.ReadWrite**;
5. parametrs – nosaka bufera izmēru
6. parametrs tipa **bool** nosaka, vai operācijas ar failu notiek sinhrona vai asinhrona režīmā.

Ir obligāti jānorāda tikai divus pirmos parametrus.

Pēc faila atvēršanas (objekta **FileStream** izveidošanas) ir pieejamas dažādas īpašības un metodes.

Īpašības **CanRead, CanWrite** un **CanSeek** apraksta plūsmas iespējas, īpašība **Length** glabā faila garumu, **Position** – plūsmas rādītāja nobīdi no plūsmas sākuma. Piemēram ar operatoru:

```
fs.Position = fs.Length;
```

var pāriet faila beigās.

Lasīšanai no faila ir paredzētas sekojošas metodes:

```
int ReadByte()
int Read(byte [] anyBuffer, int iBufferOffset, int iCount)
```

Metode **ReadByte** dod atpakaļ nolasītu no faila baitu, pārveidotu tipā **int** bez zīmes, vai –1 ja ir sasniegtas faila beigas. Metode **Read** dod atpakaļ nolasītu no faila baitu skaitu (ne vairāk kā **iCount**) vai 0, ja ir sasniegtas faila beigas. Bet parametrs **iBufferOffset** nosaka nobīdi buferī nevis plūsmā.

Piemēram, atvert failu un nolasīt tā saturu masīvā var ar operatoriem:

```
FileStream fs = new FileStream("MyFile",FileMode.Open,
                               FileAccess.Read,FileShare.Read);
Byte [] anyBuffer = new Byte[fs.Length];
```

```
fs.Read(anyBuffer, 0, (int)fs.Length);
fs.Close();
```

Informācijas rakstīšanai failā ir paredzētas metodes:

```
void WriteByte(byte byValue)
void Write(byte [] anyBuffer, int iBufferOffset, int iCount)
```

Faila rādītāja pārvietošanai var izmantot ne tikai īpašību **Position,** bet arī metodi **Seek.**

Faila aizvēršanai ir jāizmanto metodi **Close.**

Klasei **FileStream** ir viens būtisks trūkums – ar šīs klases palīdzību var ierakstīt failā vai nolasīt no faila tikai masīvus no baitiem. Bet valodā C#, atšķirība no C++ valodas, pārveidot baitu virkni pie vajadzīgā tipa ir grūti (visi vienkāršie tipi pēc savas būtības ir struktūras). Tāpēc, darbam ar teksta failiem tiek lietotas klases **StreamReader** un **SreamWriter**, bet darbam ar bināriem failiem – klases **BinaryReader** un **BinaryWriter**.

Darbs ar teksta failiem

Sakarā ar Unicode izplatīšanu, tekstveida failu apstrāde kļuva daudz sarežģītāka. Piemēram, simbola 'A' kods tabulā ASCII ir 0x41, bet Unicode sistēmā 0x0041. Tādējādi, Unicode rindas, kas satur galvenokārt ASCII simbolus satur ļoti daudz nuļļu, kas rada ļoti daudz problēmu programmām, pierakstītām valodā C ka arī programmām, pierakstītām Unix sistēmai, kas interpretē 0 kā rindas gala pazīmi.

Klases **StreamWriter** un **StreamReader** nodrošina darbu ar dažāda formāta failiem. Par rindu pārveidošanu Unicode formātā un atpakaļ atbild klases, nodefinētas vārdu telpā **System.Text.** Vissgalvenākā no tām ir klase **Encoding**.

Klases **SreamWriter** konstruktors, ar kuru palīdzību var atvērt teksta failu rakstīšanai, izskatās šādi:

```
StreamWriter(string strFileName, bool bAppend, Encoding enc)
```

Obligāti ir jānorāda tikai pirmo parametru. Ja parametrs **Encoding** nav norādīts, tad rindas tiks saglabātas formātā UTF-8. UTF-8 ta ir simbolu kodēšanas shēma, kas ir paredzēta Unicode simbolu attēlošanai bez nulles baitiem. Dotā shēmā katra Unicode simbola glabāšanai tiek izdalīti no 1 līdz 6 baitiem. Unicode simbolu no ASCII diapazona glabāšanai ir nepieciešams 1 baits. Tādējādi, Unicode rindas, kas satur tikai ASCII simbolus tiek pārveidotas ASCII failos. Ja ir nepieciešams saglabāt rindas kādā citā formātā, tad, kā trešo parametru klases **StreamWriter** konstruktoram var nodod sekojošus objektus: **Encoding.ASCII, Encoding.Unicode, Encoding.UTF7** un t.t.

Informācijas rakstīšanai failā **StreamWriter** piedāvā 17 metodes **Write** variantu, kas atļauj norādīt kā parametru jebkuru objektu, kurš tiek automātiski pārveidots rindā ar savu metodi **ToString**. Lai izvadītu failā jaunas rindas simbolu ir nepieciešams izsaukt metodi **WriteLine** bez parametriem. Bez tam eksistē metožu **Write** un **WriteLine** versijas, kas formatē rindas analoģiski metodēm **Console.Write** un **Console.WriteLine**.

Sekojošā programma katru reizi pēc palaišanas ieraksta tekstu faila beigās:

```
using System;
using System.IO;
class StreamWriterDemo
{
      public static void Main()
```

```
        {
              StreamWriter sw = new StreamWriter("StreamWriterDemo.txt", true);
              sw.WriteLine("You ran the program on {0}", DateTime.Now);
              sw.Close();
        }
}
```

Teksta failu lasīšanai ir paredzēta klase **StreamReader**. Atvērt failu ir iespējams izmantojot vienu no **StreamReader** konstruktoriem:

```
StreamReader(string strFileName, Encoding enc, bool bDetect)
StreamReader(string strFileName, bool bDetect)
```

Obligāti ir jānorāda tikai faila nosaukums. Ja parametram **bDetect** piešķirt **true**, tad konstruktors mēģinās patstāvīgi noteikt faila kodēšanas veidu. Ja abi parametri **bDetect** un **Encoding** ir norādīti, tad konstruktors mēģinās noteikt kodēšanas veidu patstāvīgi, bet ja tas neizdosies, tad tiek lietota norādītā kodēšanas sistēma.

StreamReader klase piedāvā sekojošas metodes informācijas lasīšanai:

`int Peek(), int Read()` –
dod atpakaļ sekojošā simbola kodu no faila vai –1 ja ir sasniegts faila gals.

`int Read(char [] achBuffer, int iBufferOffset, int iCount)` –
dod atpakaļ nolasītu simbolu skaitu vai 0, ja ir sasniegts faila gals.

`string ReadLine()` –
*dod atpakaļ nolasītu rindu vai **null** ja ir sasniegts faila gals.*

Piemēram:

```
StreamReader sr = new StreamReader("MyData.txt");
string str;
while ((str=sr.ReadLine())!=null)
      Console.WriteLine(str);
sr.Close();
```

Darbs ar bināriem failiem

Bināru failu apstrādei izņemot **FileStream** var izmantot klases **BinaryWriter** un **BinaryReader** objektus. Kā pirmo parametru klases **BinaryWriter** un **BinaryReader** konstruktoram ir jānodod klases **FileStream** objektu. Ja objektus **BinaryWriter** un **BinaryReader** ir plānots izmantot teksta failu apstrādei, tad kā otro parametru šo objektu konstruktoram ir ieteicams nodod **Encoding** tipa objektu**,** lai norādītu simbolu kodēšanas veidu.

Klase **BinaryWriter** piedāvā metodi **Write** informācijas rakstīšanai failā. Kā parametru metodei **Write** var nodod jebkuru bāzes tipa objektu (**bool, char, char [], string, int, float, ...).** Šī metode nesaglabā failā informāciju par datu tipu. Katram tipam tiek lietots vajadzīgais baitu skaits. Piemēram, **bool** tipa mainīgais aizņem failā 1 baitu, **float** tipa mainīgais – 4 baitus.

Klase **BinaryReader** informācijas lasīšanai piedāvā dažādākās metodes **Read[Type]**, piemēram metodes:

```
bool ReadBoolean()
byte ReadByte()
byte [] ReadBytes(int iCount)
int ReadInt32()
ushort ReadUInt16()
```

```
float ReadSingle()
un t.t.
```

Ja ir sasniegts faila gals,visas šīs metodes ģenerē izņēmumu **EndOfStreamException**, tāpēc šo metožu izsaukumus ir ieteicams ievietot blokā **try**.

Pārvietoties plūsmas **BinaryWriter** un **BinaryReader** var ar metodes **Seek** palīdzību, bet aizvērt failu, uz kura balstās **BinaryWriter** un **BinaryReader** objekti, var ar metodi **Close()**.

Darbs ar failu sistēmu

Informāciju par failu sistēmu var iegūt ar klases **Environment** palīdzību. Piemēram, ar operatoru

```
Environment.GetFolderPath(Environment.SpecialFolder.System)
```

var iegūt Windows foldera vārdu, bet ar operatoru

```
Environment.GetFolderPath(Environment.SpecialFolder.Personal)
```

tekoša lietotāja **My Documents** folderu.

Ar **Environment** klases īpašību **CurrentDirectory** var iegūt tekošā foldera nosaukumu vai izmainīt tekošo folderi. Piemēram:

```
Environment.CurrentDirectory = "D:\\";
```

Klase **Path**, kas ir aprakstīta vārdu telpā **System.IO** piedāvā daudz statisku metožu, paredzētu faila nosaukuma analīzei. Šīs metodes darbojas neatkarīgi no tā, vai fails eksistē vai nē. Piemēram, izteiksme

```
Path.GetExtention("DirA\\MyFile.txt")
```

dod atpakaļ rindu ".txt".

Izveidot jaunu folderu var ar operatoru

```
Directory.CreateDirectory("newfolder");
```

bet izdzēst folderu var ar klases **Directory** statisko metodi **Delete**, kurai kā parametrs ir jānodod ceļš pie dzēšamā foldera.

Noteikt, vai norādītais folders eksistē, var ar klases **Directory** metodi **Exists**. Bet metode **GetFiles** dod atpakaļ masīvu no rindām, kas satur visu failu vārdus, kas atrodas norādītā folderā.

Klase **File** piedāvā daudz statisku metožu darbam ar failiem un informācijas saņemšanai par tiem. Starp tām ir tādas metodes kā **Copy("Source","Destination"), Move("Source","Destination"), Delete(strFileName), GetCreationTime(strFileName)** un t.t.